Exhibit B

Content Management System (CMS) Files as of November 6, 2000

file	rev	action	time
//depot/docs/AppInstallManager-LLD.doc	1	add	9/5/2000 21:53
//depot/docs/AppInstallManager-LLD.doc	2	edit	9/5/2000 21:55
//depot/docs/AppInstallManager-LLD.doc	3	edit	9/7/2000 12:05
//depot/docs/AppInstallManager-LLD.doc	4	edit	9/12/2000 9:31
//depot/docs/Builder/AppInstallBlock-LLD.doc	1	add	8/17/2000 15:41
//depot/docs/Builder/AppInstallBlock-LLD.doc	2	edit	8/17/2000 17:48
//depot/docs/Builder/AppInstallBlock-LLD.doc	3	edit	8/17/2000 17:58
//depot/docs/Builder/AppInstallBlock-LLD.doc	4	edit	8/18/2000 11:32
//depot/docs/Builder/AppInstallBlock-LLD.doc	5	edit	8/18/2000 15:14
//depot/docs/Builder/AppInstallBlock-LLD.doc	6	edit	8/18/2000 15:33
//depot/docs/Builder/AppInstallBlock-LLD.doc	7	edit	8/22/2000 14:45
//depot/docs/Builder/AppInstallBlock-LLD.doc	8	edit	8/22/2000 15:00
//depot/docs/Builder/AppInstallBlock-LLD.doc	9	edit	8/22/2000 15:39
//depot/docs/Builder/AppInstallBlock-LLD.doc	10	edit	8/22/2000 18:46
//depot/docs/Builder/AppInstallBlock-LLD.doc	11	edit	9/5/2000 10:54
//depot/docs/Builder/AppInstallBlock-LLD.doc	12	edit	9/5/2000 11:11
//depot/docs/Builder/AppInstallBlock-LLD.doc	13	edit	9/6/2000 14:57
//depot/docs/Builder/AppInstallBlock-format.vsd	1	add	8/17/2000 15:41
//depot/docs/Builder/AppInstallBlock-format.vsd	2	edit	8/17/2000 17:48
//depot/docs/Builder/AppInstallBlock-format.vsd	3	edit	8/17/2000 17:58
//depot/docs/Builder/AppInstallBlock-format.vsd	4	edit	8/18/2000 11:32
//depot/docs/Builder/AppInstallBlock-format.vsd	5	edit	8/18/2000 15:14
//depot/docs/Builder/AppInstallBlock-format.vsd	6	edit	8/22/2000 14:45
//depot/docs/Builder/AppInstallBlock-format.vsd	7	edit	8/22/2000 15:00
//depot/docs/Builder/AppInstallBlock-format.vsd	8	edit	8/22/2000 18:46
//depot/docs/Builder/Builder-HLD.doc	1	add	8/2/2000 13:40
//depot/docs/Builder/Builder-HLD.doc	2	edit	8/3/2000 10:52
//depot/docs/Builder/Builder-HLD.doc	3	edit	8/3/2000 15:36
//depot/docs/Builder/Builder-HLD.doc	4	edit	8/4/2000 18:32
//depot/docs/Builder/Builder-HLD.doc	5	edit	8/7/2000 8:25
//depot/docs/Builder/Builder-HLD.doc	6	edit	8/7/2000 18:08
//depot/docs/Builder/Builder-HLD.doc	7	edit	8/11/2000 12:16
//depot/docs/Builder/Builder-HLD.doc	8	edit	8/16/2000 11:16
//depot/docs/Builder/Builder-HLD.doc	9	edit	8/16/2000 11:50
//depot/docs/Builder/Builder-HLD.doc	10	edit	8/16/2000 11:56
//depot/docs/Builder/Builder-HLD.doc	11	edit	8/16/2000 12:00
//depot/docs/Builder/Builder-HLD.doc	12	edit	8/16/2000 12:19
//depot/docs/Builder/Builder-HLD.doc	13	edit	8/21/2000 9:43
//depot/docs/Builder/Builder-LLD.doc	1	add	8/9/2000 10:51
//depot/docs/Builder/Builder-LLD.doc	2	edit	8/9/2000 15:59
//depot/docs/Builder/Builder-LLD.doc	3	edit	8/21/2000 10:42
//depot/docs/Builder/BuilderTrickyIssues.doc	1	add	9/6/2000 17:01
//depot/docs/Builder/BuilderUI-LLD.doc	1	add	8/28/2000 10:46
//depot/docs/Builder/BuilderUI-LLD.doc	2	edit	9/6/2000 14:27
//depot/docs/Builder/FSRFD-LLD.doc	1	add	8/23/2000 19:28
//depot/docs/Builder/FSRFD-LLD.doc	2	edit	8/25/2000 18:27
//depot/docs/Builder/FSRFD-LLD.doc	3	edit	9/1/2000 18:49
//depot/docs/Builder/FSRFD-LLD.doc	4	edit	9/7/2000 20:22
//depot/docs/Builder/Installmon-LLD.doc	1	add	8/25/2000 18:27
//depot/docs/Builder/Installmon-LLD.doc	2	edit	8/28/2000 10:46
//depot/docs/Builder/Installmon-LLD.doc	3	edit	9/5/2000 11:28

//depot/docs/Builder/builder-desc.doc	1 add	8/23/2000 13:09
//depot/docs/Builder/builder-desc.doc	2 edit	8/25/2000 16:42
//depot/docs/Builder/builder-desc.vsd	1 add	8/25/2000 16:42
//depot/docs/Builder/builder-diagram.vsd	1 add	8/4/2000 17:55
//depot/docs/Builder/builder-diagram.vsd	2 edit	8/4/2000 18:06
//depot/docs/Builder/builder-diagram.vsd	3 edit	8/7/2000 9:15
//depot/docs/Builder/builder-diagram.vsd	4 edit	8/16/2000 11:10
//depot/docs/Builder/builder-diagram.vsd	5 edit	8/16/2000 12:19
//depot/docs/Builder/builder-fam-lld.doc	1 add	8/18/2000 15:33
//depot/docs/Builder/builder-fam-lld.doc	2 edit	8/25/2000 14:16
//depot/docs/Builder/builder-fam-lld.doc	3 edit	8/30/2000 19:31
//depot/docs/Builder/builder-fam-lld.doc	4 edit	8/31/2000 10:49
//depot/docs/Builder/builder-fam-lld.doc	5 edit	9/5/2000 11:07
//depot/docs/Builder/builder-fam-lld.doc	6 edit	9/7/2000 17:45
//depot/docs/Builder/builder-package-lld.doc	1 add	8/21/2000 8:33
//depot/docs/Builder/builder-package-lld.doc	2 edit	8/25/2000 14:16
//depot/docs/Builder/builder-package-lld.doc	3 edit	9/5/2000 11:07
//depot/docs/Builder/builder-package-lld.doc	4 edit	9/6/2000 14:57
//depot/docs/Builder/builder-profile-lld.doc	1 add	8/18/2000 15:33
//depot/docs/Builder/builder-profile-lld.doc	2 edit	8/18/2000 16:21
//depot/docs/Builder/builder-profile-lld.doc	3 edit	8/18/2000 17:41
//depot/docs/Builder/builder-profile-lld.doc	4 edit	8/21/2000 8:33
//depot/docs/Builder/builder-profile-lld.doc	5 edit	8/25/2000 14:16
//depot/docs/Builder/builder-profile-lld.doc	6 edit	8/30/2000 19:31
//depot/docs/Builder/builder-profile-lld.doc	7 edit	8/31/2000 10:49
//depot/docs/Builder/builder-profile-lld.doc	8 edit	9/5/2000 11:07
//depot/docs/Builder/builder-profile-lld.doc	9 edit	9/7/2000 17:45
//depot/docs/Builder/eStreamSet-lld.doc	1 add	8/23/2000 13:09
//depot/docs/Builder/eStreamSet-lld.doc	2 edit	8/23/2000 15:13
//depot/docs/Builder/eStreamSet-lld.doc	3 edit	8/23/2000 15:19
//depot/docs/Builder/eStreamSet-lld.doc	4 edit	8/23/2000 15:23
//depot/docs/Builder/eStreamSet-lld.doc	5 edit	8/23/2000 15:35
//depot/docs/Builder/eStreamSet-lld.doc	6 edit	8/23/2000 15:47
//depot/docs/Builder/eStreamSet-lld.doc	7 edit	8/25/2000 11:08
//depot/docs/Builder/eStreamSet-lld.doc	8 edit	9/5/2000 11:08
//depot/docs/Builder/eStreamSet-lld.doc	9 edit	9/6/2000 10:56
//depot/docs/Builder/eStreamSet-lld.doc	10 edit	9/6/2000 11:08
//depot/docs/Builder/eStreamSet-lld.doc	11 edit	9/6/2000 11:36
//depot/docs/Builder/eStreamSet-lld.doc	12 edit	9/7/2000 16:19
//depot/docs/Builder/estream-set-format.vsd	1 add	8/23/2000 13:09
//depot/docs/Builder/estream-set-format.vsd	2 delete	8/23/2000 13:23
//depot/docs/Builder/estreamSet-format.vsd	1 add	8/23/2000 13:24
//depot/docs/Builder/estreamSet-format.vsd	2 edit	8/23/2000 15:13
//depot/docs/Builder/estreamSet-format.vsd	3 edit	8/23/2000 15:47
//depot/docs/Builder/estreamSet-format.vsd	4 edit	9/6/2000 10:56
//depot/docs/Builder/estreamSet-format.vsd	5 edit	9/6/2000 11:36
//depot/docs/CUILLD.doc	1 add	9/6/2000 18:35
//depot/docs/CUILLD.doc	2 delete	9/12/2000 11:32
//depot/docs/CUIStrawMan.doc	1 add	9/4/2000 15:27
//depot/docs/CUIStrawMan.doc	2 edit	9/6/2000 11:16
//depot/docs/CUIStrawMan.doc	3 edit	9/6/2000 13:57
//depot/docs/CUIStrawMan.doc	4 delete	9/12/2000 11:30

//depot/docs/Client Docs/Client Spoof Registry.doc	1 add	8/16/2000 11:05
//depot/docs/Client Docs/Client Spoof Registry.doc	2 edit	8/16/2000 11:14
//depot/docs/Client Docs/Client UI Module.doc	1 add	8/16/2000 11:06
//depot/docs/Client Docs/Client UI Module.doc	2 edit	8/16/2000 11:14
//depot/docs/Client Docs/Client UI Module.doc	3 edit	8/23/2000 11:51
//depot/docs/Client Docs/Installation Manager.doc	1 add	8/16/2000 11:10
//depot/docs/Client Docs/Installation Manager.doc	2 edit	8/16/2000 11:18
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	1 add	8/16/2000 11:11
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	2 edit	8/16/2000 13:14
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	3 edit	8/16/2000 13:33
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	4 edit	8/23/2000 11:51
//depot/docs/Client Docs/eStream Client Network Component.doc	1 add	8/16/2000 13:21
//depot/docs/Client Docs/eStream Registry Spoofer current status.doc	1 add	8/16/2000 11:09
//depot/docs/Client Install.doc	1 add	7/19/2000 10:51
//depot/docs/ClientNW-LLD.doc	1 add	9/6/2000 16:21
//depot/docs/ClientNW-LLD.doc	2 edit	9/11/2000 13:34
//depot/docs/Configuration.vsd	1 add	7/19/2000 10:51
//depot/docs/Downgrade.vsd	1 add	7/19/2000 10:51
//depot/docs/ECMLLD.doc	1 add	8/14/2000 16:19
//depot/docs/ECMLLD.doc	2 edit	8/18/2000 17:44
//depot/docs/ECMLLD.doc	3 edit	8/31/2000 15:44
//depot/docs/ECMLLD.doc	4 edit	9/8/2000 17:03
//depot/docs/ECMStrawMan.doc	1 add	8/10/2000 19:08
//depot/docs/Estream Client-Server Diagram.vsd	1 add	7/14/2000 18:31
//depot/docs/Estream High-Level Design Diagram.vsd	1 add	7/14/2000 18:31
//depot/docs/Estream High-Level Design Diagram.vsd	2 edit	7/25/2000 11:09
//depot/docs/Install a subscribed application.doc	1 add	7/18/2000 19:19
//depot/docs/Install a subscribed application.doc	2 edit	7/18/2000 20:05
//depot/docs/Install a subscribed application.doc	3 edit	7/18/2000 20:12
//depot/docs/Install.vsd	1 add	7/19/2000 10:51
//depot/docs/LSMLLD.doc	1 add	8/21/2000 20:19
//depot/docs/LSMLLD.doc	2 edit	9/5/2000 19:15
//depot/docs/LSMLLD.doc	3 delete	9/12/2000 11:26
//depot/docs/LSMStrawMan.doc	1 add	9/1/2000 13:36
//depot/docs/LSMStrawMan.doc	2 edit	9/1/2000 14:51
//depot/docs/LSMStrawMan.doc	3 edit	9/1/2000 17:11
//depot/docs/LSMStrawMan.doc	4 delete	9/12/2000 11:28
//depot/docs/License Subscription Manager (LSM).doc	1 add	8/16/2000 11:06
//depot/docs/Omnishift BasePatent.doc	1 add	7/17/2000 11:06
//depot/docs/Omnishift BasePatent.doc	2 edit	7/17/2000 14:54
//depot/docs/Omnishift BasePatent.doc	3 edit	7/17/2000 16:03
//depot/docs/Omnishift BasePatent.doc	4 edit	7/17/2000 16:38
//depot/docs/Omnishift BasePatent.doc	5 edit	7/17/2000 17:51
//depot/docs/Omnishift BasePatent.doc	6 edit	7/17/2000 18:27
//depot/docs/OmnishiftCodingStandard.doc	1 add	8/23/2000 10:16
//depot/docs/OmnishiftCodingStandard.doc	2 edit	8/25/2000 18:09
//depot/docs/OmnishiftCodingStandard.doc	3 edit	8/25/2000 18:15
//depot/docs/OmnishiftCodingStandard.doc	4 delete	9/12/2000 12:04
//depot/docs/PrefetchStrawMan.doc	1 add	9/4/2000 10:25
//depot/docs/PrefetchStrawMan.doc	2 delete	9/12/2000 11:22
//depot/docs/QUESTIONS TO ASK AN ASP.doc	1 add	8/10/2000 16:13
//depot/docs/QUESTIONS TO ASK AN ASP.doc	2 edit	8/18/2000 10:40

//depot/docs/QUESTIONS TO ASK AN ASP.doc	3	edit	8/22/2000 20:29
//depot/docs/QUESTIONS TO ASK AN ASP.doc	4	delete	9/12/2000 12:02
//depot/docs/SLiM-LLD.doc	1	add	8/21/2000 9:24
//depot/docs/SLiM-LLD.doc	2	delete	8/24/2000 8:37
//depot/docs/ServerDocs/AppServer-LLD.doc	1	add	8/28/2000 18:53
//depot/docs/ServerDocs/AppServer-LLD.doc	2	edit	8/29/2000 13:09
//depot/docs/ServerDocs/AppServer-LLD.doc	3	edit	9/5/2000 20:43
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	1	add	9/5/2000 9:33
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	2	edit	9/5/2000 10:33
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	3	edit	9/8/2000 16:46
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	4	delete	9/13/2000 17:37
//depot/docs/ServerDocs/Monitor-LLD.doc	1	add	9/5/2000 9:35
//depot/docs/ServerDocs/Monitor-LLD.doc	2	edit	9/5/2000 12:39
//depot/docs/ServerDocs/Monitor-LLD.doc	3	edit	9/8/2000 16:46
//depot/docs/ServerDocs/Monitor-LLD.doc	4	edit	9/8/2000 16:48
//depot/docs/ServerDocs/Monitor-LLD.doc	5	delete	9/13/2000 17:40
//depot/docs/ServerDocs/SLiM-LLD.doc	1	add	8/24/2000 8:36
//depot/docs/ServerDocs/SLiM-LLD.doc	2	edit	8/25/2000 17:03
//depot/docs/ServerDocs/SLiM-LLD.doc	3	edit	8/29/2000 19:23
//depot/docs/ServerDocs/SLiM-LLD.doc	4	edit	9/12/2000 12:22
//depot/docs/ServerDocs/WebServerDB-LLD.doc	1	add	8/30/2000 14:43
//depot/docs/ServerDocs/WebServerDB-LLD.doc	2	edit	9/6/2000 14:47
//depot/docs/ServerDocs/WebServerDesign.doc	1	add	8/23/2000 10:14
//depot/docs/ServerDocs/WebServerDesign.doc	2	edit	8/25/2000 17:05
//depot/docs/ServerDocs/WebServerDesign.doc	3	delete	8/30/2000 18:26
//depot/docs/ServerDocs/corba_centric.doc	1	add	8/23/2000 10:14
//depot/docs/ServerDocs/db_centric.doc	1	add	8/23/2000 10:14
//depot/docs/ServerDocs/eStreamConfigManagement.doc	1	add	8/23/2000 10:14
//depot/docs/ServerDocs/eStreamConfigManagement.doc	2	edit	9/1/2000 16:22
//depot/docs/ServerDocs/eStreamHTTP.doc	1	add	8/23/2000 10:14
//depot/docs/ServerDocs/eStreamLoggingDesign.doc	1	add	8/23/2000 10:14
//depot/docs/ServerDocs/eStreamLoggingDesign.doc	2	edit	9/1/2000 16:13
//depot/docs/Uninstalling a subscribed application.doc	1	add	7/20/2000 10:55
//depot/docs/Upgrade1.vsd	1	add	7/19/2000 10:51
//depot/docs/Upgrade2.vsd	1	add	7/19/2000 10:51
//depot/docs/Upgrade3.vsd	1	add	7/19/2000 10:51
//depot/docs/client-driver-design.htm	1	add	7/3/2000 14:02
//depot/docs/client-driver-design.htm	2	edit	7/6/2000 14:38
//depot/docs/client-driver-design.htm	3	edit	9/5/2000 21:53
//depot/docs/client_scenarios.htm	1	add	7/14/2000 17:56
//depot/docs/eStream 1.0 Functional Spec.doc	1	add	7/10/2000 18:17
//depot/docs/eStream 1.0 Functional Spec.doc	2	edit	7/10/2000 19:17
//depot/docs/eStream 1.0 Functional Spec.doc	3	edit	7/11/2000 10:01
//depot/docs/eStream 1.0 Functional Spec.doc	4	edit	7/11/2000 10:33
//depot/docs/eStream 1.0 Functional Spec.doc	5	edit	7/11/2000 14:04
//depot/docs/eStream 1.0 Requirements.doc	1	add	7/3/2000 13:58
//depot/docs/eStream 1.0 Requirements.doc	2	edit	7/10/2000 18:17
//depot/docs/eStream 1.0 Requirements.doc	3	edit	7/12/2000 14:27
//depot/docs/eStream 1.0 Requirements.doc	4	edit	7/31/2000 16:52
//depot/docs/eStream 1.0 Requirements.doc	5	delete	7/31/2000 16:58
//depot/docs/eStream 1.0 Top Level Components.doc	1	add	7/12/2000 14:12
//depot/docs/eStream1.0-HLD.doc	1	add	7/31/2000 15:43

//depot/docs/eStream1.0-HLD.doc	2 delete	9/12/2000 11:59
//depot/docs/eStream1.0-REQ.doc	1 branch	7/31/2000 16:58
//depot/docs/eStream1.0-REQ.doc	2 edit	8/8/2000 11:13
//depot/docs/eStream1.0-REQ.doc	3 edit	8/10/2000 10:54
//depot/docs/eStream1.0-REQ.doc	4 edit	8/11/2000 12:10
//depot/docs/eStream1.0-REQ.doc	5 edit	9/5/2000 9:45
//depot/docs/eStream1.0-REQ.doc	6 edit	9/6/2000 11:16
//depot/docs/eStream1.0-REQ.doc	7 edit	9/7/2000 17:33
//depot/docs/eStream1.0-REQ.doc	8 delete	9/12/2000 11:56
//depot/docs/eStream1.0-SCALE.doc	1 add	8/10/2000 13:36
//depot/docs/eStream1.0-SCALE.doc	2 edit	8/10/2000 13:47
//depot/docs/eStream1.0-SCALE.doc	3 delete	9/12/2000 12:00
//depot/docs/eStreamStrawMan.doc	1 add	8/8/2000 14:44
//depot/docs/eStreamStrawMan.doc	2 edit	8/10/2000 19:02
//depot/docs/efsd-lld.doc	1 add	8/14/2000 14:17
//depot/docs/efsd-lld.doc	2 edit	8/21/2000 9:11
//depot/docs/efsd-lld.doc	3 edit	8/23/2000 11:26
//depot/docs/efsd-lld.doc	4 edit	9/11/2000 9:58
//depot/docs/file-spoofer-lld.doc	1 add	8/22/2000 10:07
//depot/docs/osrdocs/apc.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/blue-screen.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/blue-screen_files/Image13.gif	1 add	7/6/2000 14:38
//depot/docs/osrdocs/c++-support.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/cacheman.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/cancel-1.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/cancel-2.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/converting-nt4-5.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/defensive-driv.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/driver101.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/driver101_files/arch.gif	1 add	7/6/2000 14:38
//depot/docs/osrdocs/driver101_files/setup.gif	1 add	7/6/2000 14:38
//depot/docs/osrdocs/driver102.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/fs-recognizers.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/impl-pnp.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/integ-build.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/io-completion.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/io-completion_files/iocomp1.gif	1 add	7/6/2000 14:38
//depot/docs/osrdocs/io-completion_files/iocomp2.gif	1 add	7/6/2000 14:38
//depot/docs/osrdocs/io-completion_files/iocomp3.gif	1 add	7/6/2000 14:38
//depot/docs/osrdocs/kernel-context.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/kernel-dlls.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/lanman-server.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/nt-io-requests.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/oplocks.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/rename.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/security1.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/seh.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/sync-mech.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/vm-1.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/vm-2.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/windbg-ext.html	1 add	7/6/2000 14:38
//depot/docs/osrdocs/wmi.html	1 add	7/6/2000 14:38

//depot/docs/osrdocs/work-queues.html	1 add	7/6/2000 14:38
//develop/docs/Builder/AppInstallBlock-LLD.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/AppInstallBlock-LLD.doc	2 edit	9/1/2000 18:02
//develop/docs/Builder/AppInstallBlock-LLD.doc	3 edit	9/1/2000 18:08
//develop/docs/Builder/AppInstallBlock-LLD.doc	4 edit	9/5/2000 9:37
//develop/docs/Builder/AppInstallBlock-LLD.doc	5 edit	9/5/2000 9:39
//develop/docs/Builder/AppInstallBlock-format.vsd	1 add	8/26/2000 15:53
//develop/docs/Builder/Builder-HLD.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/Builder-LLD.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/FSRFD-LLD.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/InstallMon-LLD.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/TrickyBuilderIssues.doc	1 add	9/6/2000 16:29
//develop/docs/Builder/builder-desc.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/builder-desc.vsd	1 add	8/26/2000 15:53
//develop/docs/Builder/builder-diagram.vsd	1 add	8/26/2000 15:53
//develop/docs/Builder/builder-fam-lld.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/builder-fam-lld.doc	2 edit	9/1/2000 18:02
//develop/docs/Builder/builder-fam-lld.doc	3 edit	9/1/2000 18:08
//develop/docs/Builder/builder-package-lld.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/builder-package-lld.doc	2 edit	9/1/2000 18:02
//develop/docs/Builder/builder-package-lld.doc	3 edit	9/1/2000 18:08
//develop/docs/Builder/builder-profile-lld.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/builder-profile-lld.doc	2 edit	9/1/2000 18:02
//develop/docs/Builder/builder-profile-lld.doc	3 edit	9/1/2000 18:08
//develop/docs/Builder/eStreamSet-lld.doc	1 add	8/26/2000 15:53
//develop/docs/Builder/eStreamSet-lld.doc	2 edit	9/1/2000 18:02
//develop/docs/Builder/eStreamSet-lld.doc	3 edit	9/1/2000 18:08
//develop/docs/Builder/estreamSet-format.vsd	1 add	8/26/2000 15:53
//develop/docs/Client Docs/Client Spoof Registry.doc	1 add	8/26/2000 15:53
//develop/docs/Client Docs/Client UI Module.doc	1 add	8/26/2000 15:53
//develop/docs/Client Docs/Installation Manager.doc	1 add	8/26/2000 15:53
//develop/docs/Client Docs/License Subscription Manager (LSM).doc	1 add	8/26/2000 15:53
//develop/docs/Client Docs/eStream Client Network Component.doc	1 add	8/26/2000 15:53
//develop/docs/Client Docs/eStream Registry Spoofer current status.doc	1 add	8/26/2000 15:53
//develop/docs/Client Install.doc	1 add	8/26/2000 15:53
//develop/docs/Configuration.vsd	1 add	8/26/2000 15:53
//develop/docs/Downgrade.vsd	1 add	8/26/2000 15:53
//develop/docs/ECMLLD.doc	1 add	8/26/2000 15:53
//develop/docs/ECMStrawMan.doc	1 add	8/26/2000 15:53
//develop/docs/Estream Client-Server Diagram.vsd	1 add	8/26/2000 15:53
//develop/docs/Estream High-Level Design Diagram.vsd	1 add	8/26/2000 15:53
//develop/docs/Install a subscribed application.doc	1 add	8/26/2000 15:53
//develop/docs/Install.vsd	1 add	8/26/2000 15:53
//develop/docs/LSMLLD.doc	1 add	8/26/2000 15:53
//develop/docs/License Subscription Manager (LSM).doc	1 add	8/26/2000 15:53
//develop/docs/Omnishift BasePatent.doc	1 add	8/26/2000 15:53
//develop/docs/OmnishiftCodingStandard.doc	1 add	8/26/2000 15:53
//develop/docs/QUESTIONS TO ASK AN ASP.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	1 add	8/28/2000 16:10
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	2 edit	9/1/2000 17:20
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	3 edit	9/4/2000 16:35
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	4 edit	9/5/2000 10:27

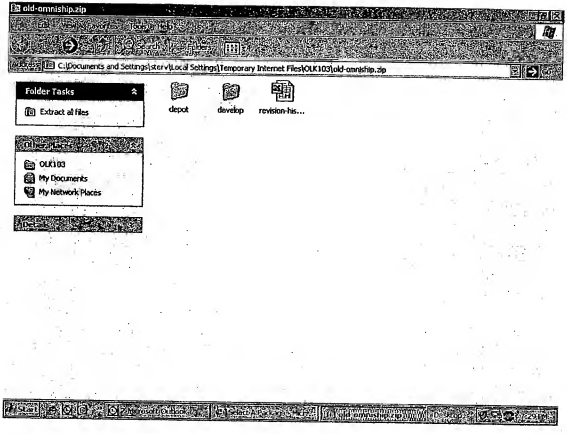
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	5 edit	9/5/2000 10:31
//develop/docs/ServerDocs/Monitor-LLD.doc	1 add	8/28/2000 16:10
//develop/docs/ServerDocs/Monitor-LLD.doc	2 edit	9/1/2000 17:20
//develop/docs/ServerDocs/Monitor-LLD.doc	3 edit	9/4/2000 16:35
//develop/docs/ServerDocs/SLiM-LLD.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/SLiM-LLD.doc	2 edit	8/30/2000 8:50
//develop/docs/ServerDocs/WebServerDesign.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/corba_centric.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/db_centric.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/eStreamConfigManagement.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/eStreamHTTP.doc	1 add	8/26/2000 15:53
//develop/docs/ServerDocs/eStreamLoggingDesign.doc	1 add	8/26/2000 15:53
//develop/docs/Uninstalling a subscribed application.doc	1 add	8/26/2000 15:53
//develop/docs/Upgrade1.vsd	1 add	8/26/2000 15:53
//develop/docs/Upgrade2.vsd	1 add	8/26/2000 15:53
//develop/docs/Upgrade3.vsd	1 add	8/26/2000 15:53
//develop/docs/client-driver-design.htm	1 add	8/26/2000 15:53
//develop/docs/client_scenarios.htm	1 add	8/26/2000 15:53
//develop/docs/eStream 1.0 Functional Spec.doc	1 add	8/26/2000 15:53
//develop/docs/eStream 1.0 Top Level Components.doc	1 add	8/26/2000 15:53
//develop/docs/eStream1.0-HLD.doc	1 add	8/26/2000 15:53
//develop/docs/eStream1.0-REQ.doc	1 add	8/26/2000 15:53
//develop/docs/eStream1.0-SCALE.doc	1 add	8/26/2000 15:53
//develop/docs/eStreamStrawMan.doc	1 add	8/26/2000 15:53
//develop/docs/efsd-lld.doc	1 add	8/26/2000 15:53
//develop/docs/file-spoofer-lld.doc	1 add	8/26/2000 15:53
//develop/docs/osrdocs/apc.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/blue-screen.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/blue-screen_files/Image13.gif	1 add	8/26/2000 15:53
//develop/docs/osrdocs/c++-support.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/cacheman.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/cancel-1.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/cancel-2.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/converting-nt4-5.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/defensive-driv.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/driver101.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/driver101_files/arch.gif	1 add	8/26/2000 15:53
//develop/docs/osrdocs/driver101_files/setup.gif	1 add	8/26/2000 15:53
//develop/docs/osrdocs/driver102.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/fs-recognizers.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/impl-pnp.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/integ-build.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/io-completion.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/io-completion_files/iocomp1.gif	1 add	8/26/2000 15:53
//develop/docs/osrdocs/io-completion_files/iocomp2.gif	1 add	8/26/2000 15:53
//develop/docs/osrdocs/io-completion_files/iocomp3.gif	1 add	8/26/2000 15:53
//develop/docs/osrdocs/kernel-context.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/kernel-dlls.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/lanman-server.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/nt-io-requests.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/oplocks.html	1 add	8/26/2000 15:53
//develop/docs/osrdocs/rename.html	1 add	8/26/2000 15:53

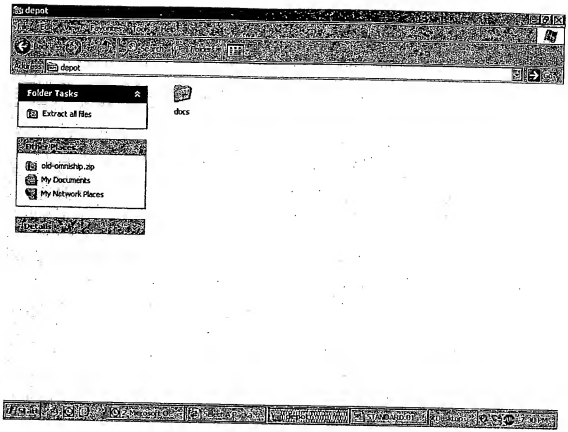
//develop/docs/osrdocs/security1.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/seh.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/sync-mech.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/vm-1.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/vm-2.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/windbg-ext.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/wmi.html	1	add	8/26/2000 15:53
//develop/docs/osrdocs/work-queues.html	1	add	8/26/2000 15:53
//develop/eng/docs/CodingStandard.doc	1	branch	9/29/2000 8:15
//develop/eng/docs/CodingStandard.doc	2	edit	10/25/2000 11:48
//develop/eng/docs/OmnishiftCodingStandard.doc	1	add	9/12/2000 15:03
//develop/eng/docs/OmnishiftCodingStandard.doc	2	edit	9/13/2000 14:44
//develop/eng/docs/OmnishiftCodingStandard.doc	3	edit	9/20/2000 11:42
//develop/eng/docs/OmnishiftCodingStandard.doc	4	delete	9/29/2000 8:16
//develop/eng/docs/SoftwareDevelopmentProcess.doc	1	add	9/29/2000 8:13
//develop/eng/docs/estream1.0-plan.xls	1	add	9/14/2000 16:26
//develop/eng/docs/estream1.0-plan.xls	2	delete	9/25/2000 11:41
//develop/eng/docs/readme.txt	1	add	9/12/2000 15:01
//develop/eng/estream/docs/HLD-builder.vsd	1	add	10/25/2000 9:12
//develop/eng/estream/docs/HLD-client.vsd	1	add	10/25/2000 9:12
//develop/eng/estream/docs/HLD-server.vsd	1	add	10/25/2000 9:12
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	3	edit	9/12/2000 12:05
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	4	edit	9/12/2000 15:03
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	5	edit	9/21/2000 16:54
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.vsd	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.vsd	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/Builder-HLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/Builder-HLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/Builder-HLD.vsd	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/Builder-HLD.vsd	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	3	edit	10/6/2000 15:36
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	4	edit	10/6/2000 17:51
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	5	edit	10/6/2000 20:05
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	6	edit	10/9/2000 18:48
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	7	edit	10/12/2000 14:21
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	3	edit	10/6/2000 15:36
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	4	edit	10/9/2000 18:48
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	5	edit	10/12/2000 14:21
//develop/eng/estream/docs/builder/BuilderTrickyIssues.doc	1	add	9/14/2000 13:21
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	3	edit	9/12/2000 15:03
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	4	edit	9/21/2000 11:28
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	5	edit	9/21/2000 13:59
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	6	edit	9/28/2000 10:20
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	7	edit	10/3/2000 16:58

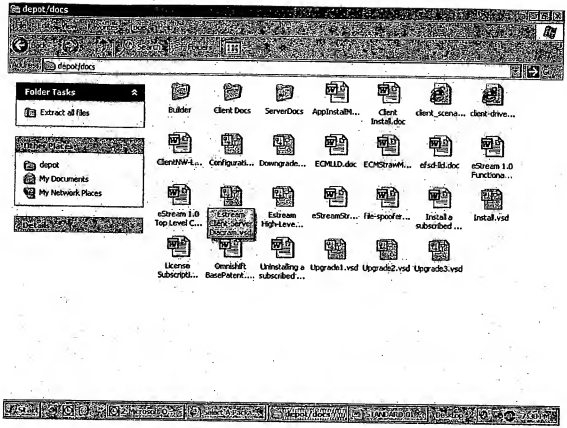
B3

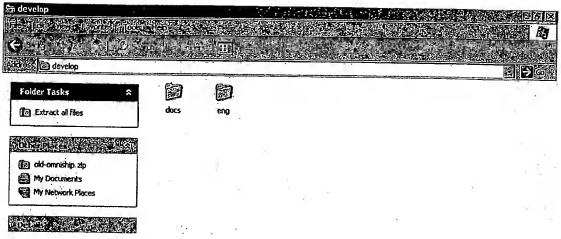
//develop/eng/estream/docs/builder/EStreamSet-LLD.vsd	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/EStreamSet-LLD.vsd	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/EStreamSet-LLD.vsd	3	edit	10/5/2000 18:49
//develop/eng/estream/docs/builder/EStreamSet-LLD.vsd	4	edit	10/6/2000 15:36
//develop/eng/estream/docs/builder/builderUI/BuilderUI-LLD.doc	1	add	9/14/2000 13:21
//develop/eng/estream/docs/builder/fsrfd/FSRFD-LLD.doc	1	add	9/14/2000 13:21
//develop/eng/estream/docs/builder/installmon/Installmon-LLD.doc	1	add	9/14/2000 13:21
//develop/eng/estream/docs/builder/packager/BuilderPackager-LLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/packager/BuilderPackager-LLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/packager/BuilderPackager-LLD.doc	3	edit	9/11/2000 16:45
//develop/eng/estream/docs/builder/profiler/BuilderFAM-LLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/profiler/BuilderFAM-LLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/profiler/BuilderProfiler-LLD.doc	1	add	9/11/2000 13:54
//develop/eng/estream/docs/builder/profiler/BuilderProfiler-LLD.doc	2	edit	9/11/2000 14:04
//develop/eng/estream/docs/builder/profiler/BuilderProfiler-LLD.doc	3	edit	9/21/2000 16:07
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	1	add	9/12/2000 9:36
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	2	edit	9/12/2000 10:19
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	3	edit	9/22/2000 16:08
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	4	edit	9/24/2000 18:56
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	5	edit	9/26/2000 22:19
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	6	edit	9/28/2000 14:01
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	7	edit	9/28/2000 20:53
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	1	add	9/20/2000 17:45
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	2	edit	9/20/2000 17:58
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	3	edit	9/21/2000 10:40
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	4	edit	9/21/2000 11:47
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	5	edit	9/22/2000 11:26
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	6	edit	9/25/2000 20:07
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	7	edit	9/26/2000 10:31
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	8	edit	9/26/2000 10:45
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	9	edit	9/26/2000 10:52
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	10	edit	9/26/2000 13:59
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	11	edit	9/26/2000 18:16
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	1	add	9/13/2000 10:34
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	2	edit	9/15/2000 17:00
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	3	edit	9/19/2000 10:17
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	4	edit	9/19/2000 21:21
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	5	edit	9/20/2000 14:23
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	6	edit	9/20/2000 17:04
//develop/eng/estream/docs/client/cni/ClientNetworking-LLD.doc	1	add	9/11/2000 13:38
//develop/eng/estream/docs/client/cni/ClientNetworking-LLD.vsd	1	add	9/11/2000 13:38
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	1	branch	9/12/2000 11:32
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	2	edit	9/13/2000 10:56
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	3	edit	9/13/2000 12:27
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	4	edit	9/13/2000 15:26
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	5	edit	9/21/2000 11:47
//develop/eng/estream/docs/client/cui/ClientUserInterface-SM.doc	1	branch	9/12/2000 11:30
//develop/eng/estream/docs/client/ecm/CacheManager-LLD.doc	1	add	9/11/2000 13:38
//develop/eng/estream/docs/client/ecm/CacheManager-LLD.vsd	1	add	9/11/2000 13:38
//develop/eng/estream/docs/client/ecm/CacheManager-SM.doc	1	add	9/11/2000 13:38
//develop/eng/estream/docs/client/efsd/efsd-lld.doc	1	add	9/19/2000 19:31
//develop/eng/estream/docs/client/epf/PrefetcherFetcher-LLD.doc	1	add	9/11/2000 20:19

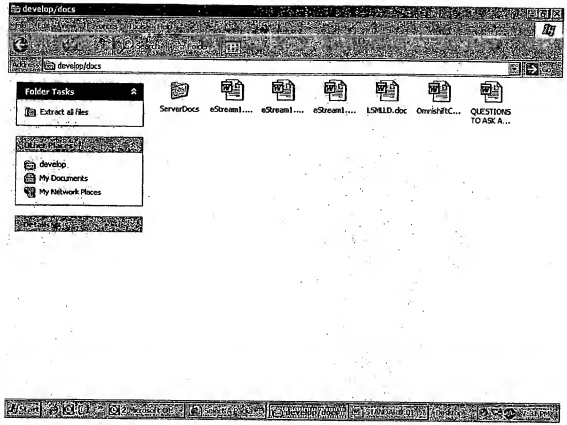
//develop/eng/estream/docs/client/epf/PrefetcherFetcher-LLD.doc	2 edit	9/21/2000 18:27
//develop/eng/estream/docs/client/epf/PrefetcherFetcher-SM.doc	1 branch	9/12/2000 11:22
//develop/eng/estream/docs/client/fsp/file-spoofer-lld.doc	1 add	9/19/2000 19:31
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	1 branch	9/12/2000 11:26
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	2 edit	9/13/2000 18:29
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	3 edit	9/13/2000 18:44
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	4 edit	9/13/2000 19:02
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	5 edit	9/13/2000 19:06
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	6 edit	9/13/2000 20:41
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	7 edit	9/13/2000 20:54
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	8 edit	9/14/2000 11:44
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	9 edit	9/21/2000 11:47
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	10 edit	9/21/2000 18:27
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-SM.doc	1 branch	9/12/2000 11:28
//develop/eng/estream/docs/development-plan.xls.xls	1 add	9/25/2000 12:57
//develop/eng/estream/docs/development-plan.xls.xls	2 edit	10/9/2000 10:51
B1 //develop/eng/estream/docs/development-plan.xls.xls	3 edit	10/11/2000 15:14
//develop/eng/estream/docs/development-plan.xls.xls	4 edit	10/19/2000 18:11
//develop/eng/estream/docs/development-plan.xls.xls	5 edit	10/26/2000 10:35
//develop/eng/estream/docs/development-plan.xls.xls	6 edit	11/1/2000 17:26
//develop/eng/estream/docs/eStream1.0-ASP.doc	1 branch	9/12/2000 12:02
//develop/eng/estream/docs/eStream1.0-ASP.doc	2 delete	9/14/2000 8:26
B4 //develop/eng/estream/docs/eStream1.0-HLD.doc	1 branch	9/12/2000 11:59
//develop/eng/estream/docs/eStream1.0-HLD.doc	2 edit	10/25/2000 9:12
//develop/eng/estream/docs/eStream1.0-Req.doc	1 branch	9/12/2000 11:56
//develop/eng/estream/docs/eStream1.0-SCALE.doc	1 branch	9/12/2000 12:00
//develop/eng/estream/docs/eStreamFS-SM.doc	1 add	9/11/2000 13:38
//develop/eng/estream/docs/eng/OmnishiftCodingStandard.doc	1 branch	9/12/2000 12:04
//develop/eng/estream/docs/eng/OmnishiftCodingStandard.doc	2 delete	9/12/2000 15:04
//develop/eng/estream/docs/mpr.doc	1 add	9/23/2000 10:43
//develop/eng/estream/docs/readme.txt	1 add	9/8/2000 17:06
//develop/eng/estream/docs/readme.txt	2 edit	9/11/2000 13:27
//develop/eng/estream/docs/server/AppServer-LLD.doc	1 add	9/22/2000 14:39
B2 //develop/eng/estream/docs/server/ComponentFramework-LLD.doc	1 branch	9/13/2000 17:37
//develop/eng/estream/docs/server/ComponentFramework-LLD.doc	2 edit	9/30/2000 12:07
//develop/eng/estream/docs/server/EMS-LLD.doc	1 add	9/13/2000 19:58
//develop/eng/estream/docs/server/EMS-LLD.doc	2 edit	9/14/2000 12:21
//develop/eng/estream/docs/server/EMS-LLD.doc	3 edit	9/14/2000 12:27
B5 //develop/eng/estream/docs/server/LoadMonApplet-LLD.doc	1 add	10/30/2000 18:12
//develop/eng/estream/docs/server/LoadMonApplet-LLD.doc	2 edit	10/31/2000 11:17
//develop/eng/estream/docs/server/Monitor-LLD.doc	1 branch	9/13/2000 17:40
//develop/eng/estream/docs/server/Monitor-LLD.doc	2 edit	9/14/2000 8:05
//develop/eng/estream/docs/server/SLiM-LLD.doc	1 add	9/12/2000 12:28
//develop/eng/estream/docs/server/Server Installation Requirements.doc	1 add	10/12/2000 9:22
//develop/eng/estream/docs/server/Server Installation Requirements.doc	2 edit	10/23/2000 11:38
//develop/eng/estream/docs/server/Server Installation Requirements.doc	3 edit	10/23/2000 16:57
//develop/eng/estream/docs/server/Server Installation Requirements.doc	4 edit	11/3/2000 12:13
//develop/eng/estream/docs/server/WebServerDB-LLD.doc	1 add	9/15/2000 9:21
//develop/eng/estream/docs/server/WebServerDB-LLD.doc	2 edit	10/30/2000 16:35
//develop/eng/estream/docs/server/WebServerDB-LLD.doc	3 edit	10/30/2000 19:12
//develop/eng/estream/docs/server/WebServerDB-LLD.doc	4 edit	11/2/2000 17:30
//develop/eng/estream/docs/wbs.mpp	1 add	9/23/2000 10:38

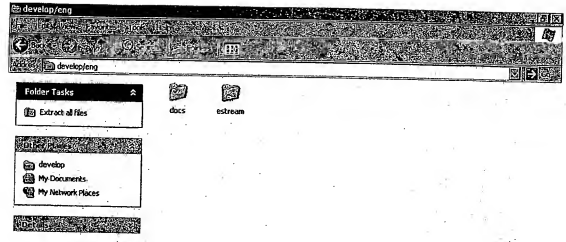

old-omnishipzip
C:\Documents and Settings\ster\Local Settings\Temporary Internet Files\OLK103\old-omnishipzip
Folder Tasks
Extract all files
OLK103
My Documents
My Network Places
depot
develop
revision-his...

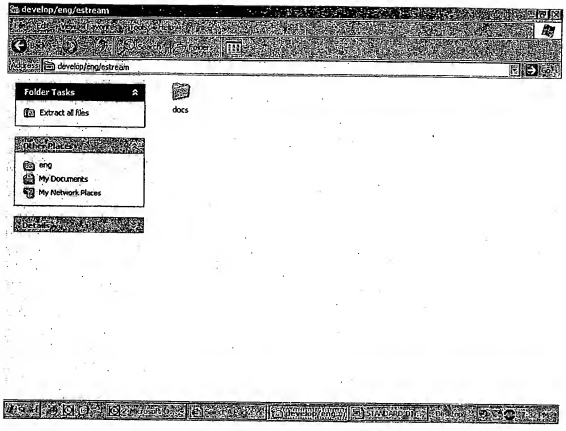

depot
depot
Folder Tasks
Extract all files
old-omnishp.zip
My Documents
My Network Places
docs

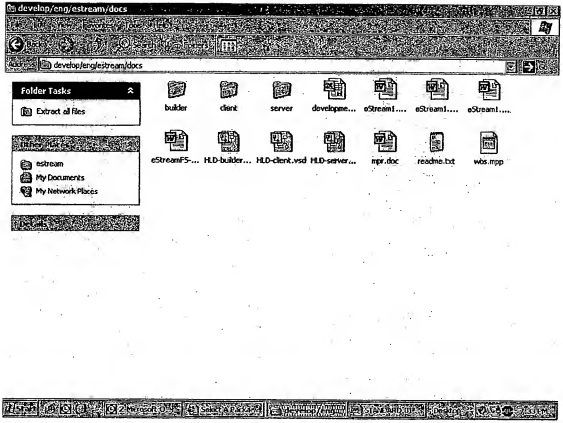

depot/docs
Folder Tasks
Extract all files
depot
My Documents
My Network Places
Builder
Client Docs
ServerDocs
AppInstallN...
Client Install.doc
client_scena...
client-drive...
ClientNW-i...
Configurati...
Downgrade...
ECMLLD.doc
ECMStrawH...
efsd-ll.doc
eStream 1.0 Functiona...
eStream 1.0 Top Level C...
eStream Client Server Description
Estream High-Leve...
eStreamStr...
file-spoofer...
Install a subscribed ...
Install.vsd
License Subscripti...
Omnishift BasePatent....
Uninstalling a subscribed ...
Upgrade1.vsd
Upgrade2.vsd
Upgrade3.vsd

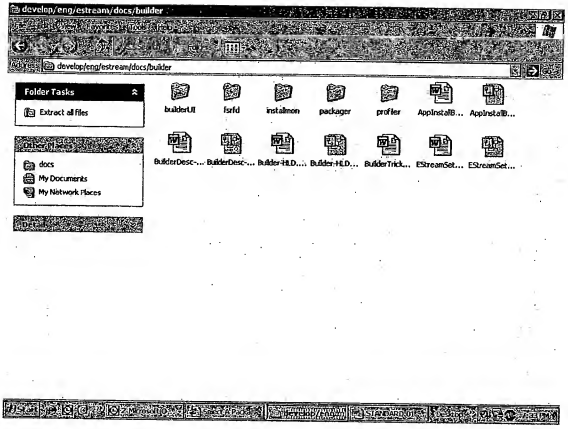

develop
develop
Folder Tasks
Extract all files
docs
eng
old-omniship.zip
My Documents
My Network Places

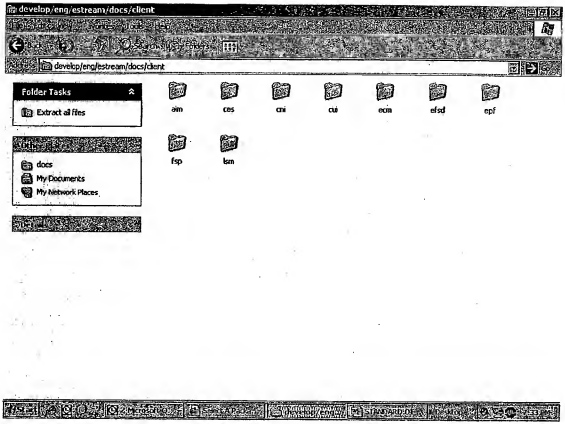

develop/docs
develop/docs
Folder Tasks
Extract all files
develop
My Documents
My Network Places
ServerDocs
eStream1....
eStream1....
eStream1....
LSMLD.doc
OmrishRC...
QUESTIONS TO ASK A...


develop/eng
develop/eng
Folder Tasks
Extract all files
develop
My Documents
My Network Places
docs
estream


develop/eng/estream
develop/eng/estream
Folder Tasks
Extract all files
eng
My Documents
My Network Places
docs


develop/eng/estream/docs
Folder Tasks
Extract all files
estream
My Documents
My Network Places
builder
client
server
developme...
eStream1....
eStream1....
eStream1....
eStreamFS-...
HLD-builder...
HLD-client.vsd
HLD-server...
mpr.doc
readme.txt
wbs.mpp


develop/eng/estream/docs/builder
develop/eng/estream/docs/builder
Folder Tasks
Extract all files
docs
My Documents
My Network Places
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD...
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...


develop/eng/estream/docs/client
develop/eng/estream/docs/client
Folder Tasks
Extract all files
docs
My Documents
My Network Places
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm

eStream Application Install Manager Low Level Design

Nicholas Ryan
Version 0.8

Functionality

The Application Install Manager (AIM) is a component of the eStream client executable. It is responsible for installing and uninstalling eStream applications at the request of the License Subscription Manager (LSM). AIM uses the information contained in an AppInstallBlock to prepare the user's system for execution of a given eStream application. It creates registry entries, copies files, and updates the file spoofing database. The user can then launch his application via a local shortcut or a shortcut on the eStream drive. Uninstallation involves undoing all changes made to the user's system by AIM during installation.

Data type definitions

This component uses the AppInstallBlock, but doesn't define it. This is defined in a low-level design document for the Builder component.

The AppInstallBlock is a binary data file with a versioned interface, basically consisting of:

- a header
- a list of files to install or send to the file spoofer
- a list of registry entries to install or remove
- a set of prefetch requests to communicate to the profile/prefetch component
- a set of initial profile data to communicate to the profile/prefetch component (post-version 1.0)
- a comment section
- an embedded DLL that can be loaded and executed for custom install needs
- a section containing a license agreement to be shown to the user

Many of the AIMsc functions take an AIBFileRef as an argument, which is an opaque pointer to the following structure:

```
typedef struct
{
   HANDLE          FileHandle;
   AIBFileHeader   FileHeader;
   AIBIndexEntry   *IndexEntries;
   LPCTSTR         AppName;
```

```
} AIBFileInfo, *pAIBFileInfo;
```

It is assumed that an external header file will be available that defines structures such as AIBFileHeader and AIBIndexEntry. For now, refer to the AppInstallBlock-LLD for how they might be defined.

Also, each application has a prefetch data file created for it an install time that is initialized with prefetch data from the AppInstallBlock. This data file is named and located as described in the Component Design section, and just consists of a non-padded list of the following structures:

```
typedef struct
{
  UINT32  FileNumber;
  UINT32  BlockNumber;

} PrefetchItem, *pPrefetchItem;
```

The following data types are used in the AIM and AIMsc interfaces:

```
typedef void *AIBFileRef;
```

Error codes that are assumed to be defined somewhere are:

```
SUCCESS (0)
ERROR_BUFFER_TOO_SMALL
```

Interface definitions

Application installation/uninstallation

There are only two functions exposed by AIM, one for application installation, and another for application uninstallation. Only the License Subscription Manager will be calling these functions.

UINT32
AIMInstallApplication(UINT8 AppId[16], LPCTSTR PathToAIB)

Parameters

AppId

[in] The application ID of the eStream application to install.

PathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to install.

Return Values

SUCCESS (0) if all the actions specified in the AppInstallBlock were performed successfully, an error code otherwise.

Comments

None.

UINT32
AIMUninstallApplication(UINT8 AppId[16])

Parameters

AppId

[in] The application ID of an existing eStream application to uninstall.

Return Values

If the specified application ID is not recognized, or the original AppInstallBlock is not found, an error code will be returned. Otherwise, AIM will make an attempt to undo all of the actions it performed while installing this application. It will return SUCCESS (0) if it undid enough of these actions so that any future installation of the same application will succeed.

Comments

None.

AIM Sub-Component Interface

Much of the functionality required by the AIM design will be useful to the Builder testing framework as well. This functionality will be treated as a sub-component within the AIM component, called AIMsc, and will export a well-defined interface. That interface is defined as follows.

UINT32
AIMscOpenAppInstallBlock(LPCTSTR PathToAIB, AIBFileRef *pAIBFile)

Parameters

PathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to open.

pAIBFile

[out] Returns a reference to an open AppInstallBlock file.

Return Values

SUCCESS (0) if the AppInstallBlock was opened successfully and validated, an error code otherwise.

Comments

The reference returned by this function can be used as a parameter to any of the other functions that take an AIBFileRef.

UINT32
AIMscCloseAppInstallBlock(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

SUCCESS (0) if the close succeeded, an error code otherwise.

Comments

None.

void
AIMscGetAIBVersion(AIBFileRef AIBFile, UINT32 *pAIBVersion)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AibVersion field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void
AIMscGetAIBAppId(AIBFileRef AIBFile, UINT8 pAIBAppId[16])

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AppId field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void
AIMscGetAIBVersionNo(AIBFileRef AIBFile, UINT32 *pAIBVersionNo)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersionNo

[out] Returns the value of the VersionNo field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

```
void
AIMscGetAIBShouldReboot(
  AIBFileRef      AIBFile,
  BOOLEAN         *pAIBShouldReboot)
```

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBShouldReboot

[out] Returns the value of the ShouldReboot flag in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

```
UINT32
AIMscGetAIBAppName(
  AIBFileRef AIBFile,
  LPTSTR   pAIBAppName,
  UINT16   *pSizeAIBAppName)
```

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBAppName

[out] The value of the ApplicationName field in the AppInstallBlock is copied into the memory pointed to by this address (it will be null terminated).

pSizeAIBAppName

[in, out] On input, should point to the size of the memory at *pAIBAppName*. On output, will point to the total bytes needed to hold the entire string if ERROR_BUFFER_TOO_SMALL is returned, otherwise is undefined.

Return Values

SUCCESS (0) if the value was successfully retrieved, ERROR_BUFFER_TOO_SMALL if the buffer is too small to hold the entire string, or another error code otherwise.

Comments

None.

UINT32
AIMscCheckAIBCompatibleOS(
AIBFileRef AIBFile,
BOOLEAN *pWasOSCompatible)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pWasOSCompatible

[out] Returns TRUE if the AppInstallBlock can be installed on the current OS, FALSE otherwise.

Return Values

SUCCESS (0) if the OS version was successfully retrieved and checked, an error code otherwise.

Comments

This function will check if the currently installed operating system and service is compatible with the specified AppInstallBlock (using the compatibility information contained in the AppInstallBlock). If not, it will display a detailed message to the user and return FALSE in *pWasOSCompatible*, otherwise it will do nothing and return TRUE in *pWasOSCompatible*.

```
UINT32
AIMscInstallAppFiles(
  AIBFileRef      AIBFile,
  HKEY            SpoofKey,
  HKEY            SpoofRefCountKey,
  LPCTSTR         InstallLogFile,
  BOOLEAN         *pIsRebootNeeded)
```

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

SpoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

SpoofRefCountKey

[in] An open handle to the registry key where file-spoofing reference counts are stored.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file copying, FALSE otherwise.

Return Values

SUCCESS (0) if all file install operations succeeded, an error code otherwise.

Comments

This function will perform the file copies and add the file spoofing entries specified in the File section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

For the sake of getting an error back to the user as soon as possible, this function will not undo file copies or spoof entry additions if it fails. **AIMscUninstallAppFiles** should be called to do so after the caller informs the user of the error.

```
UINT32
AIMscUninstallAppFiles(
  AIBFileRef      AIBFile,
  HKEY            SpoofKey,
  HKEY            SpoofRefCountKey,
  LPCTSTR         InstallLogFile,
  BOOLEAN         *pIsRebootNeeded)
```

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

SpoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

SpoofRefCountKey

[in] An open handle to the registry key where file-spoofing reference counts are stored.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file deletions, FALSE otherwise.

Return Values

SUCCESS (0) if enough of the file install operations were reversed so that re-installation will succeed and so that the system is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the file additions and remove the file spoof database entries specified in the install log file.

UINT32
AIMscInstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

Return Values

SUCCESS (0) if all variable modifications succeeded, an error code otherwise.

Comments

This function will perform the add/remove variable (i.e. registry entry) changes specified in the Variable section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

For the sake of getting an error back to the user as soon as possible, this function will not undo registry modifications if it fails. **AIMscUninstallAppVariables** should be called to do so after informing the user of the error.

UINT32
AIMscUninstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

Return Values

SUCCESS (0) if enough of the variable changes were reversed so that re-installation will succeed and so that the registry is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the add/remove variable (i.e. registry entry) changes specified in the install log file.

UINT32
AIMscInstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

PrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be created.

Return Values

SUCCESS (0) if prefetch block installation succeeded, an error code otherwise.

Comments

This function will install the prefetch information contained in the Prefetch section of the AppInstallBlock into *PrefetchFile*.

UINT32
AIMscUninstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

PrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be uninstalled.

Return Values

SUCCESS (0) if prefetch block uninstallation succeeded, an error code otherwise.

Comments

This function will remove the prefetch information stored at *PrefetchFile.*

UINT32
AIMscInstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

UINT32
AIMscUninstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

(NOT FUNCTIONAL IN ESTREAM 1.0)

UINT32
AIMscCallCustomInstall(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Install()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Install()*.

UINT32
AIMscCallCustomUninstall(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Uninstall()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Uninstall()*.

UINT32
AIMscEnforceLicenseAgreement(
AIBFileRef AIBFile,
BOOLEAN *pBUserAgreed)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

pBUserAgreed

[out] Returns TRUE if the user agreed to the license terms, FALSE otherwise.

Return Values

SUCCESS (0) if the license agreement was successfully displayed, an error code otherwise.

Comments

This function will extract the license agreement text included in the LicenseAgreement section of the AppInstallBlock and display it to the user. The user will be given the option to agree or not agree to the license (probably via a pair of buttons in a dialog).

UINT32
AIMscDisplayComment(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

SUCCESS (0) if the comment was successfully displayed, an error code otherwise.

Comments

This function will display to the user the comment included in the Comment section of the AppInstallBlock.

Component design

AIMsc does not have hard-coded knowledge regarding any of the standard registry and file locations used by AIM, which is why the functions in its interface take as inputs specifiers for filenames and base registry locations. Conversely, AIM itself has no knowledge of the internal structure of the AppInstallBlock file, which is why it must call AIMsc functions to work with such files.

Expansion is perform on registry entries and file paths containing certain variables, when they are read from the AppInstallBlock. These variables are defined in the Builder-LLD and will be recognized and expanded by AIM. (This includes file-spoof entries.)

AIM stores its data in the expected places for an eStream client component. All of the data AIM stores is user-specific, so it makes no use of the global locations defined for eStreams.

Registry keys

AIM stores its registry keys and values under:

HKEY_CURRENT_USER\SOFTWARE\Omnishift\eStream\AIM

This key will have its permissions modified so that ordinary users cannot modify the key (but the eStream client service will be given privileges so that it can do so). Here are the subkeys AIM places under this key:

"SpoofEntries"

Spoof entries are placed here. All spoofing is done globally, so there is no need to place it under an eStream-app specific key. Each value under this key is a pair of pathnames as follows:

<old-pathname> (REG_SZ)
- <spoofed-pathname>

"SpoofEntriesRefCounts"

Reference counting for spoof entries is done here. If multiple eStream apps are installed that want to spoof the same file, the entries must be ref-counted so that uninstall does not break the other apps. Each value under this key is a pair like this:

<old-pathname> (REG_DWORD)
- <ref-count>

Every value under SpoofEntries has a value under SpoofEntriesRefCounts with the same value name.

"<AppId>"

Every installed eStream app has its own subkey whose name is a string representation of its AppId, like so: "{00000000-0000-0000-0000-00000000000}". The values stored under each such key are:

AppId (REG_BINARY)
- AppId in binary form (16 bytes)

AppName (REG_SZ)
- name of the application (same as in the AppInstallBlock)

AppInstallBlockPath (REG_SZ)

- path to the AppInstallBlock for the application

AppInstallState (REG_DWORD)

- a value of 0 means app is installed, 1 means install is in progress, 2 mean uninstall is in progress.

Files

AIM stores per-user files at the following path:

(Path to the user's home directory)\Application Data\Omnishift\eStream\AIM

For each installed application, a separate data folder is created. The name of the folder is the AppId of the application in GUID ASCII format, like so: "{00000000-0000-0000-0000-00000000000}". The files stored under each such folder are:

<GUID string>-AIB.dat	- the AppInstallBlock file for the application
<GUID string>-Prefetch.dat	- the prefetch data file for the application
InstallLog.txt	- a generated log of what to do during uninstall

The Prefetch data file is simply an array of PrefetchItem structures (as described in the Data Structures section).

The InstallLog.txt is a list of undoable actions taken during installation. This log will be used during uninstall to determine which files and entries are safe to remove. Each line in the file contains one change, and is of the form:

ADDED or OVERWROTE or SPOOFED FILE "<filename>" (fully qualified)

ADDED or OVERWROTE KEY "<keyname>" (fully qualified)

ADDED or OVERWROTE VALUE "<valuename>" (fully qualified)

AIMInstallApplication Prototype

Installing an eStream application consists of the following steps:

1. Preparing for the installation
2. Displaying a license agreement to the user and having him agree to it
3. Installing all required local files and spoof entries for this app
4. Setting/removing registry entries as required
5. Initializing the profile and prefetch data for this app
6. Performing any required custom installation tasks
7. Displaying the comment to the user if required
8. Completing the installation
9. Rebooting the computer if necessary

AIM's policy is that if it encounters any fatal error during the execution of AIMInstallApplication, it will attempt to undo everything it did before returning. AIM also gracefully handles aborted installs and uninstalls.

Step 1 – Preparing for the installation

First, AIM checks if the application is already installed by looking for an AppId registry key for the specified AppId. If found, then the AppInstallInProgress registry value is checked. If it exists and is 1, the user is asked if he wants to re-install, otherwise, he is asked to restart an aborted or damaged installation. If the user says no, **AIMInstallApplication** cleans up and exits with an error.

Next, a free disk space check is performed to ensure that enough disk space is available for the install. The available free space must be at least twice the size of the AppInstallBlock to proceed.

Next, an AppId folder is created for the app (described earlier), and the AppInstallBlock file is copied to this folder. AIM then opens the AppInstallBlock using **AIMscOpenAppInstallBlock**. Then the AppId registry key is created and the four defined values created and initialized. The AppInstallState value in particular is set to 1 to indicate an install is in progress. If any of these operations fail, **AIMInstallApplication** cleans up and exits with an error.

Step 2 – Displaying the license

AIMscEnforceLicenseAgreement is called to display the license text to the user and ask for his agreement. If the functions fails or if the user's response is returned as FALSE, **AIMInstallApplication** cleans up and exits with an error.

Step 3 – Installing local files

The install log file to be used for this application is created or open and truncated. **AIMscInstallAppFiles** is called to copy the install files to the computer and to create the spoof entries specified in the AppInstallBlock. Handles to the spoof subkey and the spoof refcount subkey are opened and passed to this function, as well as a path to the newly created install.log file. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 10.

Step 4 – Modifying the registry

AIMscInstallAppVariables is called to perform the registry modifications specified in the AppInstallBlock. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 5 – Initializing profile/prefetch data

AIMscInstallAppPrefetchFile is called to create and initialize the prefetch file for this application. The file has the structure specified in the Data Structures section of this document. This function takes a path to the prefetch file to be created. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 6 – Performing custom install tasks

AIMscCallCustomInstall is called to extract the custom code .dll contained in the AppInstallBlock and to call the *Install()* function it exports. If **AIMscCallCustomInstall** fails, **AIMInstallApplication** cleans up and exits with an error.

Step 7 – Displaying a comment

AIMscDisplayComment is called to display any comment to the user contained in the appropriate section of the AppInstallblock. If this function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 8 – Completing the installation

The AppInstallInProgress registry value is set to 0 to indicate the install is complete. **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and any handles to open registry keys are also closed.

Step 9 – Rebooting the computer (if necessary)

If **AIMscInstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, no reboot is performed and the application is ready to be run. **AIMInstallApplication** exits returning SUCCESS (0).

AIMUninstallApplication Prototype

Uninstalling an eStream application consists of the following steps:

1. Preparing for the uninstallation
2. Undoing all modifications done to the registry during install
3. Undoing all file copies performed during install and removing spoof entries for this app
4. Deleting the profile/prefetch data for this application
5. Performing any required custom uninstallation tasks
6. Completing the uninstallation
7. Rebooting the computer if necessary

If the uninstallation fails for any reason, **AIMUninstallApplication** will tell the user that the uninstall has failed and that he should attempt to re-install the application before trying to uninstall again.

Step 1 – Preparing for the uninstallation

First, AIM checks if the application is already installed by looking for the AppId registry key corresponding to the specified **AppId**. If not found, then **AIMUninstallApplication** exits with an error.

Then, the AppInstallState value is set to 2 to indicate an uninstall is in progress. **AIMscOpenAppInstallBlock** is called to open the AppInstallBlock at the path specified by the AppInstallBlockPath key. If this fails, then **AIMUninstallApplication** exits with an error.

Step 2 – Undoing registry modifications

AIMscUninstallAppVariables is called to reverse the registry modifications specified in the AppInstallBlock. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 3 – Undoing file copies and removing spoof entries

AIMscUninstallAppFiles is called to delete the files copied during install and to remove the spoof entries written then. Handles to the spoof subkey and spoof refcount subkey are passed to this function, and are where the spoof entries are removed from. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 4 – Deleting profile/prefetch data

AIMscUninstallAppPrefetchFile will be called to remove the prefetch data stored for this application. Any failure is ignored.

Step 5 – Performing custom uninstall tasks

AIMscCallCustomUninstall is called to extract the custom code .dll contained in the AppInstallBlock and call the *Uninstall()* function it exports. If **AIMscCallCustomUninstall** fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 6 – Completing the uninstallation

AIMscGetAIBShouldReboot is called and the return value saved. Then **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and the AppId folder and all its contents are deleted. The AppId registry key and all its subkeys are deleted also. Any handles to open registry keys are closed. Any failures here are ignored.

Step 7 – Rebooting the computer (if necessary)

If **AIMscUninstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, the uninstallation is complete. **AIMUninstallApplication** exits returning SUCCESS (0).

AIMsc Function Prototypes

Prototypes for the AIMsc functions declared earlier are given in this section.

UINT32
AIMscOpenAppInstallBlock(LPCTSTR PathToAIB, AIBFileRef *pAIBFile)

First, the file at *PathToAIB* is opened. Then, the header is read in, header version and size is verified, and section sizes and offsets are verified. An opaque pointer to an AIBFileInfo structure is returned in the pAIBFile parameter.

UINT32
AIMscCloseAppInstallBlock(AIBFileRef AIBFile)

The file handle at ((AIBFileInfo *) AIBFile)->FileHandle, and the AIBFile structure is freed.

void
AIMscGetAIBVersion(AIBFileRef AIBFile, UINT32 *pAIBVersion)

void
AIMscGetAIBAppId(AIBFileRef AIBFile, UINT8 pAIBAppId[16])

void
AIMscGetAIBVersionNo(AIBFileRef AIBFile, UINT32 *pAIBVersionNo)

```
void
AIMscGetAIBShouldReboot(
  AIBFileRef      AIBFile,
  BOOLEAN         *pAIBShouldReboot)

UINT32
AIMscGetAIBAppName(
  AIBFileRef AIBFile,
  LPTSTR   pAIBAppName,
  UINT16   *pSizeAIBAppName)
```

These four functions are trivial. They directly return the corresponding value of the variable in ((AIBFileInfo *) AIBFile)->AIBFileHeader. (See the interface declaration for **AIMscGetAIBAppName** for details on its calling logic.)

```
UINT32
AIMscCheckAIBCompatibleOS(
  AIBFileRef        AIBFile,
  BOOLEAN           *pWasOSCompatible)
```

This function will call an API such as GetVersionEx (for Windows) to determine the currently running operating system. The OS version is then converted to a bitmask (using constants defined in an external AppInstallBlock header file) and compared with the OS and Service Pack bitmaps in ((AIBFileInfo *) AIBFile)->AIBFileHeader. If the bits are present, *pWasOSCompatible* is set to TRUE, otherwise FALSE.

```
UINT32
AIMscInstallAppFiles(
  AIBFileRef        AIBFile,
  HKEY              SpoofKey,
  HKEY              SpoofRefCountKey,
  LPCTSTR           InstallLogFile,
  BOOLEAN           *pIsRebootNeeded)
```

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the File section. If not found, an error code is returned. Otherwise, the section is parsed.

The File section is organized as a series of trees, with directories as non-leaf nodes and plain files as leaf nodes. All nodes are stored contiguously according to the pre-order traversal of the trees.

Each directory node contains the name of a single directory and a number indicating the number of children this node has. Each file node contains the file version and file name, and a flag indicating whether the file is to be spoofed or not. If so, then the last entry in the node is the spoofed pathname, otherwise it is the actual contents of the file itself.

(The actual structure types defined for these nodes are assumed to be defined in a header file external to AIM. See the AppInstallBlock-LLD for reference.)

A directory stack algorithm will be used to parse the trees and reconstruct the directory paths. Due to the complexity of the task, several helper functions are used by the algorithm to partition this work.

For every file copied or spoof entry added by the algorithm, an entry is made to the file at *InstallLogFile*. For the sake of brevity, no mention is made of the logging in the pseudocode below.

The parsing algorithm is as follows (TOS refers to the node at the top of the stack):

```
empty out directory stack
while there are nodes to read in the File section
    read a node

    if the node is a directory
        HandleDirectoryNode(node, ...)
    else
        HandleFileNode(node, ...)

    while stack is non-empty and TOS node number of children is 0
        pop directory stack
        if stack is non-empty
            decrement number of children in TOS node
```

Here is HandleDirectoryNode(node):

```
if node directory name contains Builder/AIM defined variables
    replace variable substrings with local expansions

if directory stack is empty
    if node directory name is not fully qualified
        error
    push onto directory stack an entry with:
        - node directory name
        - node number of children
    else
        push onto directory stack an entry with:
        - "TOS directory name" cat "directory name"
        - node number of children
```

Here's HandleFileNode(node, ...):

```
if node filename contains Builder/AIM defined variables
    replace variable substrings with local expansions
```

```
if directory stack is empty
    if node filename is not fully qualified
        error
    call DoFileInstall(filename, node, ...)
else
    if number of children in TOS node is <= 0
        error
    call DoFileInstall("TOS directory name" cat "filename", node, ...)
    decrement the number of children in TOS node
```

Here's how DoFileInstall(filename, node, ...) works:

```
if the file node is a spoof entry
    if filename already exists
        if existing version is earlier
            mark for spoofing
    else // filename does not exist
        create zero-length file at filename
        mark for spoofing

else // file will be copied not spoofed
    if filename already exists
        if this file is a .dll
            increment .dll shared ref count in registry
        if existing version cannot be read or existing version is earlier
            mark for copy
    else // filename does not exist
        add line to FilesLogPath file containing filename
        mark for copy

if marked for spoofing
    create spoof entry under SpoofKey
    create or update spoof refcount under SpoofRefCountKey

if marked for copy
    attempt to copy node file to client computer
    if copy fails
```

tell system to perform copy at reboot

The *pIsRebootNeeded* argument will be set to TRUE if any file copies were scheduled to happen at reboot, FALSE otherwise. Additionally, if any spoof entries were added, then an IOCTL will be sent to the spoof driver asking it to reload the spoof database.

The shared .dll reference count mentioned in the algorithm above is stored in a standard place in the Windows registry. AIM will create or increment this reference count for every non-spoofed .dll included in the AppInstallBlock (they can all be potentially shared since they will be placed outside of the eStream app directory). Each such .dll has an associated REG_DWORD value under the key at:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs

The value's name is the path to the .dll and the value's data is a integer that is the reference count for this .dll.

UINT32
AIMscUninstallAppFiles(

AIBFileRef	**AIBFile,**
HKEY	**SpoofKey,**
HKEY	**SpoofRefCountKey,**
LPCTSTR	**InstallLogFile,**
BOOLEAN	***pIsRebootNeeded)**

Currently, the *AIBFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppFiles** is simple. It iterates over the change entries contained in the log file, and undoes file copies and spoof entry additions when it is safe to do so. Here is the algorithm:

```
while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED <filename>"
        if filename is a .dll
            decrease refcount of .dll
            if refcount is 0
                mark for deletion
```

```
        else // file not a .dll
            mark for deletion

        if file is marked for deletion
            attempt to delete file
            if deletion fails
                tell system to perform deletion at reboot

    else if the entry is of the form "OVERWROTE <filename>"
        if filename is a .dll
            decrease refcount of .dll

    else if the entry is of the form "SPOOFED <filename>"
        decrease refcount of spoof entry for this filename
        if refcount is 0
            delete the spoof entry
            if 0-byte placeholder at filename still exists
                delete it
```

A failure to delete a file or to schedule its deletion will not cause **AIMscUninstallAppFiles** to fail.

```
UINT32
AIMscInstallAppVariables(
  AIBFileRefAIBFile,
  LPCTSTR InstallLogFile)
```

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Variable section. If not found, an error code is returned. Otherwise, the section is parsed.

The Variable section is organized as a series of trees, similar to how the File section is organized. There are two types of nodes, key nodes and value nodes. Registry keys can contain other keys and registry values, while registry values are just name/data pairs that are stored in keys. A registry value name is always a string, but its data can be stored as any one of a number of types.

Non-leaf nodes must be key nodes, while leaf nodes can either by key or value nodes. All nodes are stored contiguously according to the pre-order traversal of the trees.

(The actual structure types defined for these nodes are assumed to be contained in a header file external to AIM. See the AppInstallBlock-LLD for reference.)

A keyhandle algorithm will be used to parse the trees and create the registry keys and values. Due to the complexity of the task, several helper functions are used by the algorithm to partition this work.

For every key or value added by the algorithm, an entry is made to the file at *InstallLogFile*. For the sake of brevity, no mention is made of the logging in the pseudocode below.

The parsing algorithm is as follows (TOS refers to the node at the top of the stack):

```
empty out the key handle stack
while there are nodes to read in the Variable section
    read a node

    if the node is a key
        HandleKeyNode(node, ...)
    else
        HandleValueNode(node, ...)

    while the stack is non-empty and the TOS number of children is 0
        if TOS key handle is open
            close it
        pop the directory stack
        if the stack is non-empty
            decrement the number of children in TOS
```

Here is HandleKeyNode(node):

```
if the key name contains a Builder/AIM defined variable
    replace the variable substring with its local expansion

if the keyhandle stack is empty
    if the key name is not fully qualified
        error
    create key under HKCR, HKLM, etc. and save key handle
else
    if the number of children in TOS is <= 0
        error
```

create key under TOS key handle and save key handle

push onto the keyname stack an entry with:

- open key handle
- this number of children

Here's HandleValueNode(node, ...):

if the keyname stack is empty
error
else
if the number of children in TOS is <= 0
error
call DoInstallValue(TOS key handle, node, ...)
decrement the number of children in TOS

Here's how DoInstallValue(key handle, value node, ...) works:

if the value name contains a Builder/AIM defined variable
replace the variable substring with its local expansion
call SetValueEx(key handle, "value name", value type, value data, ...)

UINT32
AIMscUninstallAppVariables(
AIBFileRefAIBFile,
LPCTSTR InstallLogFile)

Currently, the *AIBFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppVariables** is simple. It iterates over the change entries contained in the log file, and undoes registry key and value additions when it is safe to do so. Here is the algorithm:

while there are change entries in the log file
read the next entry

if the entry is of the form "ADDED <keyname>"

```
        if key at keyname (fully qualified) still exists
            delete it and all subkeys and values

    else if the entry is of the form "ADDED <valuename>"
        if value at valuename (fully qualified) still exists
            delete it
```

Keys and values that were overwritten are not deleted, which is why those log file entries are not considered by the algorithm above.

A failure to delete one or more registry entries will not cause **AIMscUninstallAppFiles** to fail.

UINT32
AIMscInstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Prefetch section. If one is found, the prefetch data is read in and written out into the file at *PrefetchFile* as an array of PrefetchItem structures (this structure will change to match how the prefetch data items are represented in the AppInstallBlock-LLD). Any existing file at *PrefetchFile* is overwritten.

Next, the Prefetch component is called to set up an association between the new application and its prefetch file.

UINT32
AIMscUninstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

The file at *PrefetchFile* is deleted and the Prefetch component is called to remove the association between the app being uninstalled and the prefetch file.

UINT32
AIMscInstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

UINT32
AIMscUninstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

(NOT FUNCTIONAL IN ESTREAM 1.0)

UINT32
AIMscCallCustomInstall(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Code section. If one is found, the section is read in and written out again as a .dll library.

This library is loaded and the *Install()* function export is called (and its return value returned).

UINT32
AIMscCallCustomUninstall(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Code section. If one is found, the section is read in and written out again as a .dll library. This library is loaded and the *Uninstall()* function export is called (and its return value returned).

UINT32
AIMscEnforceLicenseAgreement(
AIBFileRef AIBFile,
BOOLEAN *pBUserAgreed)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the LicenseAgreement section. If one is found, the license text is read in and displayed to the user in a dialog. The user will be asked to either agree or disagree with the license, and *pBUserAgreed* will reflect his decision.

UINT32
AIMscDisplayComment(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Comment section. If one is found, the comment text is read in and displayed to the user in a dialog.

Testing design

Unit testing plans

AIM will be tested by a program that generates AppInstallBlocks designed to stress the component. AIM will be asked to install the given AIB and if successful, the resulting state of the system will be compared to the expected state had all the files and variables been installed correctly. An uninstall will then be performed and the system state also checked.

The focus of the testing will obviously be on the File and Variable sections. The other sections such as Code and Comments will be stressed also, but their boundary conditions are much simpler.

AIM's ability to gracefully handle aborted installs and uninstalls will also be tested.

Stress testing plans

The program described above can be deliberately tuned to create AppInstallBlocks of unusual size and organization. For example, AppInstallBlocks with thousands of files and registry entries, or files and entries with unusually long names, etc.

Coverage testing plans

In addition to the stress testing, deliberately malformed AppInstallBlocks will be generated by the test program to hit as much error-handling code as possible. AIM's data files and registry entries can also be deliberately mangled to help achieve this effect.

Cross-component testing plans

As soon as they are available, Builder-generated AppInstallBlocks will be tested to verify that the AIM is compatible with the Builder's output. As soon as the LSM and a browser plugin are available, the communication path from browser to LSM to AIM will be tested. As soon as the file spoofer is available, compatibility with the file spoof entries that AIM makes will be tested.

Upgrading/Supportability/Deployment design

AIM will make use of the eStream logging facility to record information about errors and other unusual conditions that occur. The log file will be useful for diagnosing problems that occur during testing and in real world situations.

If the AIM component is upgraded, it must still be able to uninstall any eStream applications installed at the time of upgrade. This entails being able to interpret old AIM registry entries and data files, including the AppInstallBlock. This is more a concern for future designers of the AIM component, however.

Open Issues

- How will the various anti-piracy strategies being considered affect the design of AIM, if at all?

Client Senarios - Install, Upgrade and Configure

Version 1.0

[redacted]

Note that these sequences do not include the various places at which the installer should stop and ask for input from the user. It is intended to describe the installation and upgrade process from a technical standpoint. The UI must be described elsewhere.

Note that not all decision points are described. I ignore ones that don't have a material impact on the scenarios at hand. For example, when I say that the user downloads the eStream client installation program from the ASP's web site, this does not preclude the installer being delivered on other media, such as via ftp or physical media. It also includes the case of the system administrator acting on behalf of the user. When I say the client installer is unpacked to the user's disk, this covers any mechanism by which the user might run the installer, such as from a network share.

The packaging of the eStream client software installer is platform-dependent. On Windows, the installer will most likely be distributed as a .ZIP file or a self-extracting executable. On Unix platforms, the installer will probably be a .RPM or a gzipped tar archive.

Scenario 1- Install eStream client SW, no previous installation

This scenario is for the normal installation process. The machine does not have eStream installed on it. Client installation should only be performed once per machine (assuming that the client is not uninstalled.)

0. (Not shown on diagram.) ASP makes eStream client software available on its web site. The client SW install package may be generic, or it may be pre-customized for the ASP with settings for the ASP's servers.

1. The user downloads the eStream client install application from an ASP via a web browser.

2. The installer is copied to the user's hard drive and extracted. (Optionally, the installer can be accessed directly from a network share, as is the likely case in a corporate intranet.)

3. The installer is run the by the user (or by the system administrator on behalf of the user.) The installer queries the registry to determine if the eStream client has already been installed on this machine. Since this is a new installation, it will find that the registry keys for the eStream client are undefined. If there are registry settings for other installed software that may influence what the eStream installer will do, those are checked at this step as well. Note that the installer may choose different actions or to install different things depending on the version of the operating system that is present.

4. Just to be sure, the installer checks to see if any of the eStream client drivers are already installed on this system. (Note that we thus need some mechanism for determining which of our drivers is installed on a client system, and the versions of these drivers. Two possibilities are to look for the files on disk or to attempt to access the loaded drivers.) Since this is a new installation, none of the drivers will be on the system.

5. The installer copies the eStream drivers to the appropriate places on the user's system so that they can be loaded on the next system reboot. (If the drivers can be installed without rebooting the system, this is preferred.)

6. Client user-mode components are installed on the client system, in a user-specified location. The browser plugin (if we provide one) is installed for the user's default browser, or optionally, for any other supported browser.

7 System files (the registry and possibly other files) are modified so that the drivers will be loaded on the next system reboot. System files are modified so that the user-level components will be started on boot or logon, if that has been requested. Default configuration information (either specified by the user, or as customized by the ASP) will be written into the appropriate eStream configuration files, or into the registry. Uninstall information is written into the appropriate place.

8. Initial (empty) versions of the cache, application registry information, and application spoof information are created or installed. Alternatively, the client software could know how to create these files if they are not found when the software is started.

9. (Not shown on diagram.) The drivers are loaded into the running system, or the system is rebooted.

Needed APIs

None specifically needed.

Scenario 2 - Upgrade eStream client SW (trivial case - same or newer version installed)

This scenario covers the case when the user attempts an install or an upgrade of the eStream client components, when the version already installed is at least as new as the version they are attempting to install. This may occur if a user does not know that eStream is already installed on a machine, or if they download and run the installer to upgrade to the latest version, when they are already running the latest version. The installer or upgrader should determine that there is no point in doing an upgrade and gracefully exit. Note that Scenario 5 covers the case where the user elects to force a reinstall or a downgrade of the client software.

0. (Not shown.) ASP provides an eStream client software install or upgrade program via their web site.

1. User downloads the eStream client software from the ASP web server. The installation or upgrade program is for a version no newer than the eStream software already installed on the client system.

2. The install or upgrade SW is extracted to the client machine's hard disk.

3. The install or upgrade software checks the registry to see if there is an installed version of the client at least as new as the installer. There is, so the installer notifies the user that a newer version is already installed. The user elects not to reinstall the client software, and the installer exits.

Needed APIs

The installer or upgrader must determine the version of the client software currently installed. Most likely, this will be done through the registry, so no APIs are specifically needed.

Scenario 3 - Upgrade eStream client SW (easy case - no kernel components needed)

This scenario covers the case where the user installs an update to the eStream client software, but only the user-mode components are newer than those installed on the client system. In this case, it should be (theoretically) possible to replace only the user-mode components, and restart eStream without rebooting. This scenario is a special case of Scenario 4. We probably want to eliminate this scenario and always install fresh copies of all client system components (other than configuration related files).

0. (Not shown.) ASP provides an eStream client software install or upgrade program via their web site.

1. User downloads the eStream client software from the ASP web server. EStream is already installed on the client's machine.

2. The install or upgrade SW is extracted to the client machine's hard disk.

3. The install or upgrade software checks the registry, and finds that the installed client software is older than that provided by the installer. It also discovers that only user-mode components need to be replaced.

4. The installer checks to see if the user-mode client components are running, and if so, if any bits are currently being served through them. If so, it brings up a dialog box asking the user to shut down any applications that may currently be accessing the z: drive. The upgrade may be cancelled at this point. Kernel components are notified that user-mode components will be coming down for an upgrade. All non-kernel client components are shut down.

5. Client user-mode components are replaced, as necessary.

6. The registry is updated with information about the newly installed client components. Any necessary changes to the uninstall information are made.

7. Any persistent data (things in the cache, configuration files, etc.) that have changed format are converted to the new format, or discarded. (Alternatively, the client software could understand both the old and the new data format, and perform the conversion itself the next time it is run.)

8. (Optional). User-mode components are restarted. No reboot is necessary.

Needed APIs

The Cache Manager must support a stop client API:

bool StopClient()

StopClient would return TRUE if the client has been stopped, and FALSE otherwise (perhaps because the user is currently running an eStreamed app, and doesn't want to stop right now.)

Scenario 4 - Upgrade eStream client SW (most general case - kernel and user components)

This scenario covers the case when kernel-mode as well as user-mode components must be replaced. I didn't create a separate scenario for the case where kernel-mode components must be replaced, but user-mode components don't need to be, since that case is really no simpler than this one.

0. (Not shown.) ASP provides an eStream client software install or upgrade program via their web site.

1. User downloads the eStream client software from the ASP web server.

2. The install or upgrade SW is extracted to the client machine's hard disk.

3. The install or upgrade software checks the registry, and finds that the installed client software is older than that provided by the installer. It discovers that at least one kernel-mode component must be replaced.

4. The installer checks to see if the user-mode client components are running. If necessary, it brings up a dialog box asking the user to shut down any applications that may currently be accessing the z: drive. The upgrade may be cancelled at this point.

Kernel-mode components are notified that eStream is being taken down for an upgrade. All non-kernel client components are shut down, and drivers are unloaded, if possible.

5. Client user-mode components are replaced, as necessary. Kernel-mode components are also replaced. It may be necessary that new versions of the drivers are placed in a special location, to be installed on the next reboot.

6. The registry and other configuration files are updated with information about the newly installed client components. Any necessary changes to the uninstall information are made.

7. Any persistent data (things in the cache, configuration files, etc.) that have changed format are converted to the new format, or discarded. (Alternatively, the client software could understand both the old and the new data formats, and perform the conversion itself.)

8. (Not shown). Machine is rebooted, either immediately or some time later. On reboot, the new kernel-mode drivers are installed in the appropriate locations and loaded.

Needed APIs

See StopClient() above.

Scenario 5 - Forced Reinstall/Downgrade of client SW

Though we would normally prefer a user to uninstall eStream before installing an older version, forced reinstalls (and possibly downgrades) may be necessary for a variety of reasons. The most important situation in which we would want to support this is when the uninstaller fails for some reason, leaving the system in a partially-installed state. Note that users often need to reinstall software because configuration files or registry settings have become corrupted, leading to aberrant application behavior. Note that downgrades are not strictly safe, because newer versions may have made incompatible changes to persistent file formats. We should thus support a reinstall mode that replaces settings files with fresh, new ones.

0. (Not shown.) The ASP provides an eStream client installer on their web site.

1. User downloads the eStream client SW installer.

2. Installer is unpacked onto the client system's hard drive.

3. The installer queries the registry and determines that it is not newer than the version of the client software already installed. The user specifically requests a reinstall or downgrade.

4. The installer checks to see if the cache manager is running. If it is, it asks it to shut itself and all other user-mode components down. If necessary, the user will be prompted to shut down any applications currently accessing the z: drive.

5. If the cache manager was running, it notifies the kernel components that the user-mode components are shutting down for replacement.

6. (Not shown on diagram.) User components exit.

7. User-mode components are all reinstalled, possibly with an older version.

8. Kernel-mode components are all reinstalled, possibly with an older version.

9. Application setup information is overwritten with default values, including empty versions of persistent caches, app install information, and app spoof information. Optionally, the user could request that the configuration information and cache not be overwritten.

10. (Not shown.) The machine is rebooted immediately or later, and the new drivers are installed upon reboot.

Needed APIs

See StopClient() above.

EStream Client SW Configuration

We have not fully defined what aspects of the client software will be configurable. Certainly, cache size and location should be configurable.

0. (Not shown.) The user brings up the eStream configuration UI via some mechanism.

1. The UI determines the current configuration and settings by querying the cache manager. If necessary, the UI starts the cache manager.

2. The cache manager queries the running user-mode components and kernel-mode components to determine the current state of the system.

3. Some of the queries may end up referencing on-disk configuration information through the client file manager.

4. If the user changes a setting, the change is sent to the cache manager to take effect immediately.

5. Notices are sent to user and kernel mode components to take effect immediately.

6. Persistent changes are written to configuration files or the registry via the client file manager.

Needed APIs

We need APIs for the client UI to get and set program settings, and APIs to read and write these settings to persistent configuration files. The exact format of this APIs is a bit fuzzy now, since we don't know all of the things that we want to configure.

eStream Client Scenarios

Introduction

The following are scenarios or tasks we've identified that will take place on a client machine. Some of the following scenarios are explicitly requested by a user; others take place in the background without a user's knowledge.

Some working definitions and assumptions here:

- The eStream file system will be referred to as the Z: drive
- A "user" is registered with an ASP. An "account" is a "billable unit" with an ASP.
- For a given ASP, there's a many-to-many relationship between users and accounts (single user is a member of multiple accounts; an account is used by multiple users).
- A "user identifier" is cached info on a client machine for a user. It identifies
  - user, password
  - ASP
  - account server
  - DRM server

nstall/upgrade eStream client SW

he first time eStream is installed, this is an explicit action by the user or a sysadmin. For upgrades, this could be xplicit, or the eStream client could detect that it needs to upgrade itself (possibly by asking permission first). Some ubcategories:

- Install is requested, latest version of eStream already installed
- User doesn't have sufficient permission to install

onfiguration of eStream client SW

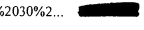
ither during installation, or subsequent to this, some client parameters need access.

- Size, location of on-disk cache
- Startup of eStream client: auto vs. manual

tart eStream client SW

his allows the client to query the account server for new subscriptions, to authenticate and start eStream'ed apps, to start ıy spoofers on the system, etc.

ıe client can start either explicitly by the user, or implicitly at boot or login.



Manage "user" and "account" data

Some subcategories:

Remove user data from an ASP

Add/remove account that this user is a member of

Subscribe/unsubscribe to an application

This can come from "within" eStream or not. I.e., the client SW may not be informed that a new subscription has been made (e.g., it's been done on the ASP web site); hence the subscribed app cannot be immediately installed.

Account/user queries

- billing info
- subscription info

"Install" account information on the client

This means to cache the "user identifier" on the client machine. This allows the eStream client do to asynchronous queries for new subscriptions, transparently start subscribed apps, etc.

Installing account info is inherently an explicitly requested action. We identified two possible scenarios:

- using a dialog to identify and "log in" to an ASP account server
- export an existing "user identifier" to a file that could be transferred to another machine, and used by the eStream client to install the same identifier on this second machine (e.g., by double-clicking on it)

"Install" a subscribed application

This means: download all necessary bits to make a subscribed app "ready-to-run."

This can be explicit (e.g., when a user subscribes to a new app) or implicit (e.g., the client SW pings the account server to look for a new subscription).

Query Z:\ (i.e., the root directory of the eStream FS)

We shouldn't require authentification if only the top-level components of the root directory are queried. Ideally, the client SW simply identifies the contents with some representation of the actual directories -- e.g., with a special icon displayed in Explorer.

Query Z:\.* (i.e., a file/directory under the root directory)

This includes actually running applications!

Accessing actual application directory contents:

- May need to authenticate before granting access
  - authentication may fail
- May not need to authenticate; e.g.,
  - already granted within time slice

- already using files within the application hierarchy

Aim Explorer at a folder with a file that's associated with an eStream'ed app

Termination of an application

- Is this the last open file associated with an application hierarchy?
- Do we need to upload user data?
- Do we need to contact DRM server to give up authentication token?

Uninstalling components

Uninstalling a subscribed app

Unclear how clean we need to leave the system

Uninstalling eStream client components

Unclear how clean we need to leave the system

Failures/errors

App crashes

eStream crashes

Client machine crashes

Kill a zombie connection

Unexpected loss of connectivity

eStream™ Client FSD Design

version 0.2, 5 [redacted]

Curt Wohlgemuth
Omnishift Technologies, Inc.
Company confidential

Introduction

This is an initial requirements/design document for the eStream file system driver (EFSD), implementing the eStream file system (EFS), for a Windows 2000 client machine.

Very high-level view

On the client side of the eStream product, the user will have a networked drive visible and accessible; this drive will "contain" all applications available in the current user's session. There will be limitations on what will be visible to a user within this drive, and there are uncertainties as to how the entire client design will lay out. However, it's still useful to specify some known issues and suggestions for how the EFSD should be designed.

Here's the general overview of what will reside on the client and what the control flow will be.

- The eStream drive will be managed by the eStream FSD, which is a form of network redirector. It will interface with the I/O manager, Cache manager, and VM manager.
- Requests for file and directory contents will be passed from the EFSD to what I'm calling the "eStream Client Manager," or ECM, which may reside in either user or kernel space.
- The ECM will handle, at least:
  - passing necessary read/write data and metadata requests to the server
  - caching all pertinent data and metadata
  - performing any profiling and prefetch work
  - helping the eStream FSD keep coherence with the server
- The network interface between the ECM and the server is the subject of other design documents

Although in practice it may be that much of the eStream drive will be read-only from a user's perspective, the EFSD must be designed for full write capabilities. Policies for read/write access will be the responsibility of the ECM and the server.

Design overview

Interfacing with the client OS

The W2000 EFSD will handle all appropriate requests from the various Windows Executive components. Here is a list of requests it needs to handle:

- Create IRPs, for both new and existing files
- Cleanup, Close IRPs
- Read and Write IRPs:
  - synchronous and asynchronous
  - cached and non-cached
  - paging and non-paging
- Fast I/O reads and writes (with buffers or MDLs)

- File information (get and set) IRPs
- Directory query IRPs
- Volume information (get and set) IRPs
- File system information (get and set) IRPs
- Various Device control IRPs (as needed by our implementation)
- Flush buffer IRPs
- System shutdown IRPs
- Various Fast I/O queries

In particular, I expect we do **not** need to handle Directory Notification IRPs, though this is potentially an open issue.

I'm also proposing we not support hard links on a Windows client (these are supported natively on NTFS on W2000 only).

The question of Byte-lock IRPs is an open one. These would probably be expensive to properly support, but it's unclear a this time if not supporting them will be a showstopper for us or not.

Interfacing with the ECM

How the eStream FSD will interface with the ECM will depend on whether the ECM is a user or kernel space service. Nevertheless, the data that needs to be exchanged between the two components can be described today.

The EFSD and the ECM need to communicate all the basic information above, including:

- Create requests
- Open requests
- Read/write requests
- Directory content requests
- Rename requests
- Delete requests
- File/directory metadata requests
- Buffer flush requests
- Volume allocation information requests (possibly)

In addition, the ECM must be able to inform the EFSD that a file or directory it has open is either:

- not being modified by another process on any system; or
- potentially being modified by another process on another system

This is necessary for the EFSD to allow caching of the file on the client machine.

What's cached where?

Due to the requirements of the NT executive components, the EFSD needs to cache various items of file metadata in its own data structures. As long as it's tracking some attributes, it makes sense to keep track of all that it easily can. In particular, the eStream FSD will keep track of:

- Time stamps: creation, last write, last access, last change other than write
- File sizes: allocated, actual (aka EOF)
- File attributes: e.g., normal, directory, read-only, hidden

The EFSD will not cache directory contents information; this will be the task of the ECM. Given how frequent these requests come through to the file system on Windows, we will definitely need to cache this on the client side.

eStream FSD design points

In no particular order, here are some items that must be implemented in the eStream FSD.

1. There are no volumes, and no VPB for a network redirector; I've verified this with the LanManager redirector.
   a. We do need to understand how we can have the EFSD handle multiple mounted drives if needed, and how to distinguish them by name in the Create IRP.

   b. We don't have to support any operations on a volume in EFS.

2. We should disallow the creation of paging files in EFS. There is a bit available for a Create IRP that specifies this, and we can complete the IRP with an unimplemented return code.
3. All file synchronization will be on an FCB basis, using the standard Resource and PagingIoResource ERESOURCE objects used by the rest of the Windows Executive.
   a. User requests will be synchronized by acquiring the main Resource -- shared for reads, exclusive for writes, other changes, deletion, etc.
   b. Paging I/O requests will be synchronized by acquiring the PagingIoResource -- shared for reads, exclusive for writes. Also exclusive access will be needed to set file sizes.
4. Most disk file systems have a resource associated with a VCB, which is acquired exclusively for creation/deletion etc. We will have a global EFS resource for this, since there are no VCBs.
5. Asynchronous requests will be handled by posting the IRP to the CriticalWorkQueue, and marking the IRP as pending.
   a. A common worker routine will be used for all async posts, which will dispatch the IRP to the appropriate real IRP routine when it's invoked.
   b. An async request will be defined as one that IoIsOperationSynchronous() returns false, and the EFSD is the top-level component (see below)
6. The EFSD will track the top-level IRP for the thread whose context it is running in. In particular,
   a. No async processing request will be honored unless the EFSD is the top-level component
   b. No cache manager requests will be made unless the EFSD is the top-level component
   c. The top-level component is expected to acquire the appropriate resources, and the EFSD must not try to acquire them as well if it is not top-level
   d. EOF file size will not be extended or changed by paging I/O
7. EFS will not support holes in files, and hence the ValidDataLength FCB field will be set to disable this.
8. Most fast I/O routines will be supported in EFS. We should be able to use the FSRTL supplied routines for fast reads and writes.
9. All cache manager resource acquire/release callbacks will be supported. All will point to common routines that simply acquire or release the main Resource for the FCB. The Context pointer passed into all of them will be the FCB for the stream.
10. Synchronous read/write requests will update the CurrentByteOffset in the File object
11. Each Create will result in a unique CCB data structure; this will be small, and only hold those few fields needed:
    a. For the Directory Control IRP, a CCB needs to hold the current entry index and the pattern originally used -- for subsequent queries
    b. A field for various flags
12. A single FCB will represent all current open stream instances of a file. When a new file is opened, the EFSD will search the current open FCBs to find one matching this file/directory name.
    a. For now, this will be a simple linked list with a linear search. We can improve this as needed
    b. The EFS global resource must be acquired exclusively:
       i. before the global FCB "list" is searched
       ii. before a new FCB is added to the list
       iii. before an FCB is deleted from the list
13. EFS will not support open by file ID; hence the FileInternalInformation class for a File Information IRP will not be supported.
14. Actual I/O will be directed to standard routines in a separate file, so they can be isolated and updated easily as our method of transferring data changes.

15. Here's how to do file/directory renames:
    a. The I/O manager will send to the EFSD this sequence:
        i. Create for source
        ii. Create for target, with the SL_OPEN_TARGET_DIRECTORY flag set
        iii. Set Information with a Rename request for the source, sending the target directory FileObject handle, and the target filename in the file info record.
    b. EFSD needs to do this:
        i. When it receives the Create for the target and the target *directory* exists, return STATUS_SUCCESS, and change the name in the FileObject to the basename of the target (the full pathname of the target is sent in), and set the Status.Information to FILE EXISTS or FILE DOES NOT EXIST, as appropriate. If the target directory doesn't even exist, return PATH NOT FOUND.
        ii. When it receives the Set Info request, if all the flags check out (e.g., if the file exists, ReplaceExisting must be TRUE), send a Rename request to the ECM.
16. Reads and writes to only regular files will be supported, not to directories.
17. Any code that touches user buffers or can call routines that may throw exceptions must be guarded by a try/except block.
18. Some tips on memory allocation (from osrdocs/defensive-driv.html)
    a. Use our own memory allocation/deallocation routines, instead of ExAllocatePool() et al. directly
    b. These routines can do various checks for trashing memory:
        i. fill allocated memory with a pre-defined bit pattern, instead of zeroes; fill deallocated memory with a different pattern.
        ii. allocate a header/trailer with standard information, like where allocated, from what pool, etc.
        iii. change the bit pattern in the header/trailer on deallocation, and look for freeing memory twice

Design outline

The EFSD will look a lot like the sample FSD from Rajeev Nagar's NT FS book, which looks a whole lot like the FASTFAT FSD source from the IFS kit.

Most IRPs will have essentially two routines each to handle them: a dispatch routine which is sent the IRP directly, and a real routine that does the actual processing. The dispatch routine is essentially a stub that calls the real one. The real one is used to handle async processing of the IRP by a worker thread.

DriverEntry

This does a whole slew of initialization, including the dispatch table, fast I/O table, the cache callbacks, the FCB list and its synchronization object, creates the FS device object, and sets up the interface with the ECM.

Create

There is one Create routine; there will be no async processing of Create requests. Ultimately, its job is to send a create or open request to the ECM, and return SUCCESS or not to its caller. It also does at least the following:

- The global FCB data structure synchronization object must be acquired exclusively
- Paging files are not allowed
- Write-through requests result in no caching
- Related file objects must be for a directory
- An absolute pathname will be generated if a related file object is specified
- It will request to open/create the file from the ECM, and if successful, get the attributes it needs for the FCB
- The FCB must be searched for by name in the global data structure; if found, this is used, else a new one created.
- A new CCB must be created for this file
- If open existing entry is specified, and not found, the correct error must be returned (e.g., FILE NOT FOUND vs. PATH NOT FOUND)
- If OPEN_TARGET_DIRECTORY is specified, EFSD must request an open for the target directory from ECM, and

confirm its existence. The FileObject's name (from the IRP) must also be munged as specified above.
- The Share Access value sent in must be checked against the existing one for the FCB if this is not the first stream associated with this FCB.
- If any error is found, all data structures allocated must be released.
- The IsFastIoPossible member is set to FastIoIsPossible.

Cleanup

There may be async posting of Cleanup requests. Some points:

- The global resource and the FCB Resource must be acquired exclusively, posting if necessary
- This represents the end of processing for this stream, but not a close of the file.
- If caching is on, the cache must be flushed, and the pages purged.
- The count of open handles in the FCB must be decremented
- Any time stamps must be updated if accesses were done using fast I/O.
- The FO_CLEANUP_COMPLETE flag in the FileObject must be set
- IoRemoveShareAccess() must be called

Close

There may be async posting of Close requests.

- The global resource must be acquired exclusively
- The CCB must be deallocated
- The file's time stamps must be updated if the CCB bits indicate access/modification
- If this is the last reference to the FCB for the file, the FCB is deallocated
  - any updated file attributes must be pushed back to the ECM in this case
- If the file is marked to be deleted on close, the ECM must be asked to delete the file

Read

Reads will definitely be open to async posting. Some points:

- Non-buffered reads need to go straight to the ECM to request data; buffered reads must request data from the cache.
- For now, we'll use the standard copy interface for cache reads
  - CcReadMdl() for MDL reads
  - CcCopyRead() otherwise
    - **Note:** the buffer to use might be for an allocated MDL, or it might be the UserBuffer!

- If this is non-paging, non-buffered, and the file stream is also open for buffering, need to:
  - grab the PagingIoResource exclusive
  - purge the cache for the range of the FCB
  - release the PagingIoResource
- Synchronization: grab the main Resource shared for non-paging reads; else the PagingIoResource shared.
- Zero length reads always return success
- Reads starting beyond EOF return EOF
- Requested lengths will be truncated to size if the request goes beyond EOF
- The cache map must be initialized if this is the first buffered read for the FCB
- For synchronous, non-paging reads, update the CurrentByteOffset in the CCB
- For non-paging reads, a bit in the CCB must be set to indicate that the file was accessed

Write

Writes will definitely be open to async posting. Some points:

- Non-buffered and buffered writes are as above for reads.
- If this is a buffered write, need to call CcCanIWrite(); if false, call CcDeferWrite() and post for async processing.
- A non-paging, non-buffered write for which the file stream is also open for buffering means:
  - grab the PagingIoResource exclusive
  - purge the cache for the range of the FCB
  - flush the cache as well **(Note: additional reqt!)**
  - release the PagingIoResource
- Writes of length zero immediately succeed
- A byte length of Low == FILE_WRITE_TO_END_OF_FILE, High == 0xffffffff signifies to start at EOF
- For paging writes:
  - No write requests beyond EOF will be honored
  - If the starting offset is EOF or beyond, just return success
  - If ending offset is beyond EOF, truncate write length to EOF
- For buffered writes:
  - If the write will extend the file size, inform the cache manager about the size change
  - If this is an MDL write, use CcPrepareMdlWrite(), else use CcCopyWrite().
  - Again, as for reads, even for a non-MDL write, there may be an MDL allocated that we should use.
- For synchronous, non-paging writes, update the CurrentByteOffset in the CCB
- For non-paging writes, set a bit in the CCB specifying that the file has been modified.

Fast I/O Read

Initially at least, we'll just set the fast I/O read routine to FsRtlCopyRead().

Fast I/O Write

Initially at least, we'll just set the fast I/O write routine to FsRtlCopyWrite().

Fast I/O Query Basic Info

This will just fill in the basic info buffer with the data in the FCB.

Fast I/O Query Standard Info

This will just fill in the standard info buffer with the data in the FCB.

Fast I/O Query Open

This will just fill in the network open info buffer with the data in the FCB, if the file exists. Some empirical observations 've made using NTFS:

- Regardless of whether the file exists or not, this will return TRUE (all fast I/O routines are boolean)
- If the file does not exist, it will set the EOF size in the buffer to 0. The AllocationSize must be non-zero. All other fields seem to be don't cares.
- If the file exists but is zero length, then both EOF and AllocationSize will be 0.
- The IRP sent to this routine is for an IRP_MJ_CREATE; we can use more than just the name to identify the file, but also the security characteristics or whatever else is sent in the IRP.

File Query Info

Standard queries will be supported; these however will not:

- FileInternalInformation -- no OPEN_BY_FILE_ID
- FileEaInformation -- no EA data
- FileCompressionInformation -- no on-disk compression
- FileStreamInformation -- no multiple streams

File Set Info

These actions will be supported:

- EndOfFile size changes
- Time stamp changes
- File position changes
- File disposition changes -- delete pending
- File rename requests

In other words, all standard file set requests will be honored except AllocationSize changes.

Directory Query

This is an ugly NT interface, and unfortunately one that we need to expose across the ECM interface as well. Some points:

- These requests come in from the I/O Manager in a context-sensitive sequence. I.e., a request will come for the initial N directory entries; the next request will be for the next M entries; etc. Kind of like strtok().
- Thus, state must be maintained from request to request. This state will be kept in the CCB for a file stream, and consists of:
  - Pattern sent in on first request
  - Index of n'th entry to start retrieving with
- My experience is that the INDEX_SPECIFIED flag is **never** set in a directory control query, even on queries subsequent to the initial one.
- Here's the algorithm as best I can grok it from the fastfat sources:
  - If the CCB pattern field is empty, and the CCB flag specifying "match all" isn't set, this is the initial query
  - For the initial query, the main FCB Resource must be acquired exclusive, else we must acquire it shared. **WHY? If we're only modifying the CCB, why lock the FCB?**
  - The user has specified an index if the SL_INDEX_SPECIFIED IRP flag is set; we're ignoring all but the first match if SL_RETURN_SINGLE_ENTRY is set.
  - If this is the initial query, parse the pattern. If "*", set the "match all" flag in the CCB. In any case, save the pattern in the CCB.
  - Start with the current index in the CCB; for the initial query, this is 0; for subsequent queries, this is the index saved in the CCB. Both of these are overruled by the SL_INDEX_SPECIFIED flag and value.
  - Both are also overruled if the SL_RESTART_SCAN flag is set; index goes back to 0.
  - After filling the values for an entry into the supplied buffer, update the CCB index field.
  - Fill the input buffer with as many entries as possible. If we run out of space, stop, and return STATUS_SUCCESS.
  - If there isn't space for the very first entry for this query (base length of record + filename), return STATUS_BUFFER_OVERFLOW.
  - Total number of bytes written to user buffer is returned in Status.Information.
  - If there are no entries in the initial query, return STATUS_NO_SUCH_FILE.
  - If there are no entries in subsequent queries, return STATUS_NO_MORE_FILES.
  - The FileIndex field of the records returned in the buffer must be fixed up to be the byte offset from this record to the next record in the buffer. The FileIndex of the last record returned in the buffer should be 0; however, it appears from the fastfat sources that this isn't the case; they update the FileIndex value to the next entry offset, even if this can't fit in the buffer! This actually makes it a lot easier.
- In order to satisfy this EFS interface, we need a relatively similar interface between the EFSD and the ECM.

- I'm proposing that only a **single** directory entry at a time be transferred between ECM and EFS.
- We can further simplify things by essentially delivering only a FileBothDirectoryInformation buffer from ECM to EFS. This is a superset of anything the I/O Manager will request of EFS.
- *Someone* has to do pattern matching: either EFS, ECM, or the eStream server. It shouldn't be the server. I'm proposing it be ECM.

- Hence, the interface from EFS to ECM for directory queries could be:
  - Input:
    - directory handle
    - pattern
    - starting index
    - buffer
    - buffer length
  - Output:
    - (filled up buffer)
    - number of bytes filled in to buffer
    - if # bytes returned is 0, return code indicates either:
      - no entries available
      - an entry is available, but the buffer isn't large enough

- This should evenly divide the work on the client machine between the EFS and the ECM, and allow the ECM code to be more portable than if all the complexity were pushed onto it.

File System Query Info

Empirically, I've noticed that the LanMan redirector returns failure for most of these requests. So, except for any user-defined FSCTL requests we want to define, I'm going to fail all of these until it turns out we need to do otherwise.

File System Set Info

Ditto for this IRP type too.

Volume Query Info

We at least need to minimally implement these requests:

- FileFsAttributeInformation
- FileFsVolumeInformation
- FileFsDeviceInformation

Volume Set Info

We will fail all requests of this type.

Flush Buffers

A buffer flush request for a file stream will mean the following:

- If the file stream isn't buffered, return immediately
- The FCB main Resource is acquired exclusive
- The Cache Manager is told to flush the buffer for the byte range of the file
- The resource is released

A buffer flush for a directory is a successful NOP.

System Shutdown

IOCTLs

eStream Client Networking Low Level Design

Dan Arai
Version1.6

Functionality

The Client Network Interface (CNI) provides the interfaces for sending messages to servers and provides threads for receiving responses and dispatching them appropriately. It uses the eStream Messaging Service (EMS) APIs to send and receive various messages to and from the application servers and SLiM servers.

The number of threads in the CNI will depend on the functionality available from the EMS. In particular, more threads are necessary if the EMS provides asynchronous messaging capability (and the CNI uses this interface). The interfaces presented by the CNI are identical for both cases, but the internal organization of the component is not.

The prefetcher will make calls to client networking interfaces (indirectly through ECM-ReservePage) to send requests for pages. Similarly, the LSM will make calls to acquire access tokens and subscription information.

The networking component is responsible for examining the stream of requests to it and deciding when to coalesce multiple page requests into a single request to the server.

The EMS does not provide reliability in the event of server failure. The CNI is responsible for handling server failover and reissuing failed requests on different servers. The CNI abstracts the servers from other parts of the system. Clients of the CNI don't need to specify a particular server to make a request.

Since the client networking component is where timeouts and retries occur, it is the component that controls the policies for how long we wait for a connection to time out and how many times we retry a request before giving up. These parameters will be tunable. Any other parameters of the CNI that make sense to tune will be tunable.

The CNI is also the component responsible for implementing the server selection policy.

Data type definitions

The CNI uses the request structure defined by the ECM.

The CNI maintains an internal queue of messages that must be sent to servers. This queue is not exposed outside of the CNI. Like the ECM request queue, this queue will be maintained as a circular, doubly-linked list.

```
typedef struct _NWRequest
{
     NWRequestType type;
     union {} parameters; /* params, depends on type */
     struct _NWRequest *next;
     struct _NWRequest *prev;
} NWRequest;

typedef enum
{
     CNI_PAGE_READ,
     CNI_ACQUIRE_ACCESS_TOKEN,
     CNI_GET_LATEST_APP_INFO,
     CNI_RENEW_ACCESS_TOKEN,
     CNI_RELEASE_ACCESS_TOKEN,
     CNI_REFRESH_APP_SERVER_SET,
     CNI_GET_SUBSCRIPTION_LIST
} NWRequestType;
```

The CNI provides an enumeration of the parameters that can be tuned. This enumeration is expected to grow as the number of tunable parameters grows.

```
typedef enum
{
     CNI_NUM_RETRIES,
     CNI_TIMEOUT,
     CNI_PROXY_ADDRESS,
     CNI_EFFECTIVE_BANDWIDTH
} NWTunableParameter;
```

Related Components

The prefetcher and LSM call on the CNI to send requests to the app and SLiM servers. The CNI makes calls to the ECM and LSM to inform them of responses that have come back from the server. The CNI will also make calls to EFSD interface functions when pages come back that satisfy EFSD requests.

Interface definitions

CNIGetPage

eStreamStatus CNIGetPage(
IN ApplicationID *app*,
IN EStreamPageNumber *page*
);

CNIGetPage is the interface used by the ECM function **ECMReservePage** to request that a page be sent by the server. (**ECMReservePage** is called indirectly by the pre-

fetcher.) Note that no distinction is made between prefetches and demand fetches. To prevent race conditions or deadlock, the requested pages must already be marked as "in flight" in the index, and any requests for these pages from the EFSD must already be on the "in flight" queue before calling this interface.

The CNI is responsible for selecting a server to direct this request to, and resending in the event of network or server failure. It will coalesce requests for multiple pages from the same application into a single request to the server.

CNIGetSubscriptionList

eStreamStatus CNIGetSubscriptionList(
IN string *Username,*
IN string *Password*
);

CNIGetSubscriptionList enqueues a request to acquire a subscription list from a SLiM server. When the subscription list is returned by the server, the client response thread will notify the LSM of the returned data via a callback defined in the LSM document.

CNIGetLatestApplicationInfo

eStreamStatus CNIGetLatestApplicationInfo(
IN uint128 *SubscriptionID*
);

CNIGetLatestApplication enqueues a request to get the latest application information for a particular app. When the server returns the result, the CNI will notify the LSM of the returned data via a callback defined in the LSM document.

CNIAcquireAccessToken

eStreamStatus CNIAcquireAccessToken(
IN uint128 *SubscriptionID,*
IN string *Username,*
IN string *Password*
);

CNIAcquireAccessToken will cause the CNI to contact a SLiM server to retrieve an access token. The CNI is responsible for issuing retries if no response is received for a request. The CNI will call the appropriate LSM callback function when the data come back.

CNIRenewAccessToken

eStreamStatus CNIRenewAccessToken(
IN AccessToken *Token,*
IN string *Username,*
IN string *Password*
);

CNIRenewAccessToken will enqueue a request for access token renewal. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIReleaseAccessToken

eStreamStatus CNIReleaseAccessToken(
IN AccessToken *Token*,
IN string *Username*,
IN string *Password*
);

CNIReleaseAccessToken will enqueue a request for releasing an access token. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIRefreshAppServerSet

eStreamStatus CNIRefreshAppServer(
IN AccessToken *Token*,
IN uint32 *BadQOS*,
IN uint32 *NoService*
);

CNIRefreshAppServerSet will enqueue a request for refreshing the app server set. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

The client networking component will also have routines for getting and setting tunable parameters.

CNISetParameter

eStreamStatus CNISetParameter(
IN NWTunableParameter *type*,
IN void **value*
);

CNISetParameter sets a parameter. The actual type of *value* is determined by *type*.

CNIGetParameter

eStreamStatus CNIGetParameter(
IN NWTunableParameter *type*,
OUT void **value*
);

CNIGetParameter queries the current value of a parameter. The actual type of *value* is determined by *type*.

Component design

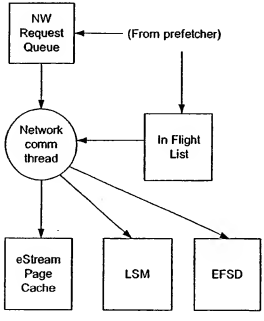
The internal organization of the client networking depends on the mechanisms available from EMS. Internally, the CNI interface functions put requests on a queue, and one or more threads services these requests by using the EMS to send messages to servers.

Synchronous Server Calls

If EMS only provides a synchronous messaging service, a single thread will be used to perform all necessary actions. The CNI interfaces will put appropriate requests on the network request queue. They will also wake up the network communication thread, if necessary.

The network communication thread's job is relatively simple. When it wakes up, it performs the following tasks:
- choose a set of requests to be coalesced and remove these from the request queue
- retrieve a server set via LSMGetAppServerSet or LSMGetSLiMServerSet, and choose a particular server for this request
- make a synchronous EMS call to send the request
- dispatch the response to the appropriate LSM or ECM callback

If the synchronous messaging mechanism becomes a performance bottleneck, we can have multiple network communication threads to increase concurrency.



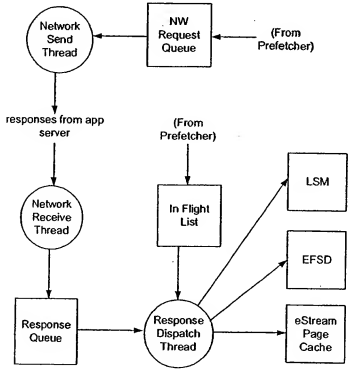
Asynchronous Server Calls

The asynchronous case is a little bit more complex. Because of the proposed asynchronous call architecture, the client NW requires three threads. The CNI interfaces work just as they do in the synchronous case. They put requests on the network request queue, and wake up the network send thread. However, the actions performed by the CNI's worker threads differ in the asynchronous model.

The network send thread is periodically awoken, and it coalesces requests off the NW request queue and sends them to the server. Unlike in the synchronous model, this thread does not synchronously wait for the request to come back from the server. Instead, it simply sends requests until the queue is empty, then goes back to sleep.

The network receive thread waits for responses to come back from any server. Because of the EMS's asynchronous call implementation details, this thread posts returned data to a queue of responses to be handled by another thread. The network receive thread is also responsible for handling timeouts and reissuing those network requests on different servers.

Finally, the response dispatch thread pulls responses off the response queue, and handles the work of dispatching them appropriately.



Handling Network Failure

When the client networking component is notified of a message failure by the EMS, the client worker thread will attempt to reissue the request on a different server.

Coalescing Multiple Requests

The CNI will coalesce multiple page requests that come from the LSM into a single request to an application server. Multiple pages requests for the same application may be coalesced. No other types of requests may be coalesced, including page requests for dif-

ferent applcations. The CNI will not produce requests larger than the maximum allowed by the application server.

Handling Persistent Failures
There will be some persistent failures that will result in the network being unable to fulfill page requests in a timely fashion. This may be due to network or server failure. (These may be indistinguishable from the CNI's point of view.) When the CNI has failed to satisfy a request for a certain amount of time, it will need to ask the user if he wants it to continue retrying, or if it should let the application terminate. It will do this via the **CUIAskUserYesNo()** interface. The client software control panel should include an option to always wait until the server is available, and never ask the user if he wants the application terminated.

Testing design

Unit testing plans

The testing harness for the networking component will be a set of dummy EMS drivers and a dummy NW client. The dummy EMS driver will be capable of performing a variety of actions, including returning appropriate responses, returning inappropriate responses, and timing out without any response. The dummy NW client will have knowledge about the expected EMS behavior, and will verify that the data it gets back from the network component are as expected.

Stress testing plans

Failure testing plans

The client NW is the sole component responsible for implementing server failover. In order to test this code, it is necessary to implement a server with predefined bad behavior. The server failure modes that must be tested include
- server that accepts a connection on a socket but doesn't respond to any requests
- server that closes the socket before sending a response
- server that closes the socket in the middle of a response
- server that sends a partial response and then just stops
- server that satisfies n requests then closes the socket or refuses to service more

It is important that we cover scenarios that look like network failures and ones that look like server failures. (Are there other failure modes that are interesting?)

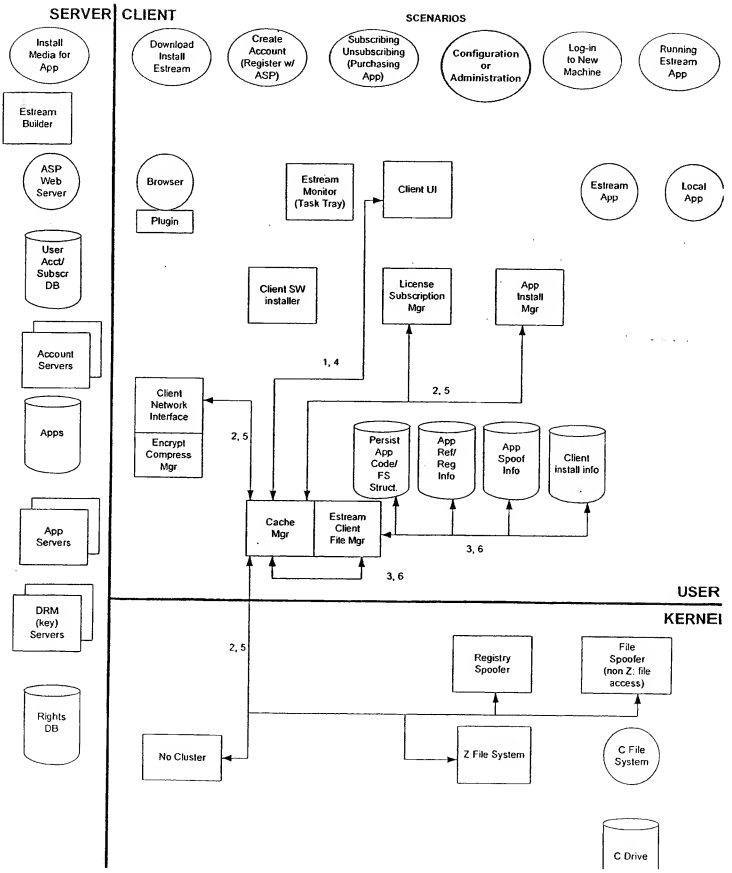
Cross-component testing plans

Cross-component testing of the client NW includes integration testing with the EMS, the LSM, and the prefetcher. Testing with the EMS can be performed in a manner similar to unit testing in conjunction with a specially written server. Testing with the LSM or pre-

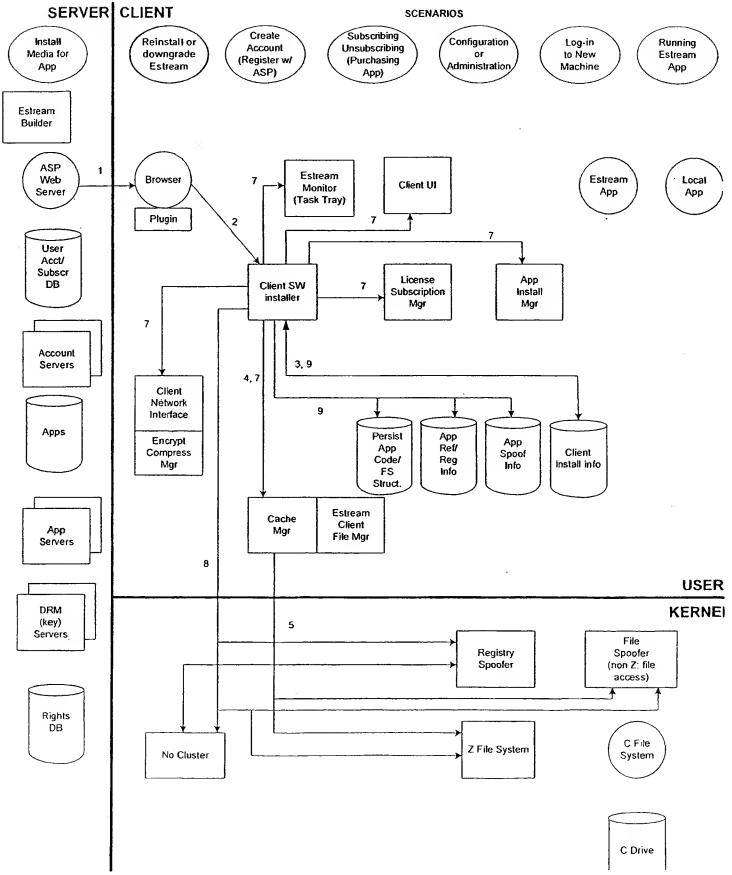
fetcher can be performed in isolation by writing drivers for either the LSM or prefetcher, and using a dummy or real EMS. I'm not sure if this sort of testing is worth the effort to write the appropriate harnesses. Verifying the output of such a combined system is certainly trickier than testing any component in isolation.

Open Issues

eStream Client Configuration



Scenario 5: eStream client SW downgrade/reinstall



eStream 1.0 Cache Manager Low Level Design

Version 1.4

[redacted]

Omnishift Technologies, Inc.
Company Confidential

Functionality

The eStream cache manager implements much of the client-side functionality for handling the eStream file system. The cache manager handles all file system requests made by the operating system by reading information from the cache or by passing the requests along to the profiling and prefetching component to fetch missing data from the network.

The cache manager will initially be implemented in user space, but it may be useful to migrate it to the kernel for improved performance. In user space, it will be part of the eStream client process. In the kernel, it will probably be a device driver distinct from the eStream file system driver.

The cache manager manages the on-disk cache of file system data, and the in-memory data structures for managing this cache. It does not manage prefetching of data from the server; that is the role of the eStream Profiling and Fetching (EPF) component. A separate networking component handles the network traffic. This component will also be described separately.

Since there is no overall discussion of the client architecture at a more detailed level than the high level design, this document will cover that as well.

Multiple cache page files will be supported. Each cache page file may be up to 2 GB in size. Different cache files may reside on different or the same logical disk (i.e. Windows drive letter.)

Data type definitions

An application ID uniquely identifies an eStream application. Just what constitutes "one" eStream application is not entirely defined, but different "builds" of the "same" app will be considered different eStream applications. For example, the Chinese-language version of Office is a different eStream application than the English-language version.

```
typedef uint128 ApplicationID;
```

The eStream page number is the data type used to describe a page number within a particular file. Note that this is a page offset, not a byte offset. For eStream 1.0, the cache manager will only support 2 GB cache files.

```
typedef uint32 EStreamPageNumber;
```

The fileId is used to uniquely identify a file within the universe of all eStream files across all eStream applications.

```
typedef struct {
     ApplicationID App,
     int32 File
} fileId;
```

The eStream page size is the fundamental size for eStream requests. This size is in bytes.

```
#define ESTREAM_PAGE_SIZE 4096
```

The eStream file system uses the file time format of the Windows operating system. If the client runs on a system with a different native time format, the client software will be responsible for translating between the native format and the eStream format. The Windows data format is a 64-bit counter of the number of 100-nanosecond periods since January 1, 1601.

EStream metadata is the file information supported by the eStream file system. This metadata is independent of the client or server operating system.

```
typedef struct
{
     uint64 CreationTime;
     uint64 AccessTime;
     uint32 FileSize;
     uint32 FileSystemAttributes;
     uint32 EStreamAttributes;
} Metadata;
```

The eStream inode contains the layout of a file in the cache. Each inode has the following structure:

```
typedef struct
{
     FileId Id; /* ID of this file; search parent for
name*/
     Metadata Metadata;
     FileID Parent; /* parent directory's file id */
     uint32 NumPages;
     PageInfo *Pages;
} EStreamInode;
```

The PageInfo array is variable sized. There is one entry in the pages array for each page in the file (not for each page cached, since we need to know whether the pages are present or not...). Note that the inode is only used in the "robust" implementation.

```
typedef struct
{
     EStreamPageNumber CachePageNumber;
     PageStatus Status;
     unsigned char Priority;
     PageChecksum Checksum;
} PageInfo;
```

The page number doesn't require the 32 bits, since pages are 4096 bytes long. The extra bits will be used to encode which cache file this page resides in. The priority field is a number representing this page's priority for being kicked out of the cache. How exactly this field is used hasn't yet been determined. The checksum is a (fast) page checksum that can be used to validate the contents of this page. Note that it will be useful to have a slower, more effective checksum for development and a faster (but less thorough) checksum for deployment.

The page status is an enumeration for the page's locking status (these are described in more detail later:

```
typedef enum
{
     PS_INVALID,
     PS_CLEAN_UNLOCKED,
     PS_CLEAN_LOCKED,
     PS_DIRTY_UNLOCKED,
     PS_DIRTY_LOCKED,
     PS_IN_FLIGHT
} PageStatus;
```

Note that this describes the layout of the tables in memory; how these data structures are represented on disk is described later.

The EFSD file handle is a small integer passed between the EFSD and the ECM. This is used opaquely by the EFSD and is used as an index into an open file table by the ECM.

```
typedef uint32 EFSDFileHandle;
```

The ECM request type specifies the request type to the rest of the system. Note that some "requests" are used to inform the prefetcher about the events handled solely by the ECM, and do not actually request that any particular action be taken by the prefetcher.

```
typedef enum
{
     ERT_READ,
     ERT_WRITE,
     ERT_READ_HIT,
```

```
    ERT_WRITE_HIT
} ECMRequestType;
```

The ECM request is a request descriptor that is used in various lists within the cache manager. These lists are doubly-linked, circular lists.

```
typedef struct _ECMRequest
{
    uint32 RequestID; /* same as EFSD request id */
    ECMRequestType RequestType;
    union {} Parameters; /* union of all parameters*/
    struct _ECMRequest *next;
    struct _ECMRequest *prev;
} ECMRequest;
```

The cache manager must maintain an array of files that have currently been opened by the EFSD. This array will be statically allocated. This will put a limit on the number of files that may be opened concurrently on the eStream file system. The elements of the array are the following:

```
typedef struct
{
    uint32 Valid;
    fileId File;
    HANDLE OpenFile; /* for simple implementation */
    eStreamInode *Inode; /* for robust implementation */
} OpenFileInfo;
```

The cache manager maintains a hash table containing information about each application that currently has open files. The hash table is indexed by app ID, and contains the following active app information records:

```
typedef struct
{
    AppID App; /* identity of this app */
    uint32 OpenFiles; /* # of open files */
    uint32 HaveAccessToken; /* boolean */
} ActiveAppInfo;
```

The ECM will use this table to quickly determine whether it should continue processing a request it gets from the EFSD, or if the request should be passed to the LSM to ensure that an access token is available. See the section below on ECM-LSM interaction for more details.

The LSM uses the access token state to specify a state for an access token. Right now, we only plan to support valid and invalid, but it may be interesting in the future to allow already opened files to be read, but no new files to be opened.

```
typedef enum
{
    ATS_INVALID,
    ATS_VALID,
    ATS_VALID_NO_OPEN
} AppTokenState;
```

Interface definitions

The ECM exports the following interfaces for operating on the cache. They may be called by the cache manager, prefetcher, or networking component. (Not all components are expected to call all interfaces; see each interface description for more details.)

Note that the cache interfaces are defined at a very high level as the actions that may be performed on the cache by the components, such as enqueuing a new request. They have been defined this way so that these intrinsic operations can be implemented correctly once and limit the possibility that an individual component will not perform proper actions.

ECMReservePage

eStreamStatus ECMReservePage(
    IN fileId *File*,
    IN EStreamPageNumber *Page*,
    IN ECMRequest **Request*
);

ECMReservePage reserves a page in the cache for a request. This interface is called by the prefetching component, and will send a request to the network component. Logically, this interface reserves an empty cache page for this request (if one is available), puts this request on the "in flight" queue, and calls on the network to request the page (unless it is already in flight.)

ECMIsPageInCache

eStreamStatus ECMIsPageInCache(
    IN fileId *File*,
    IN EStreamPageNumber *Page*
);

ECMIsPageInCache returns TRUE if the specified block is in the cache, and FALSE otherwise. It is used by the EPF to determine if it should prefetch a block; normally, the EPF would choose not to prefetch something that is already in the cache. Note that it would be a good idea for the prefetcher to adjust the priority of a page that it thinks it wants to prefetch, so that they are less likely to be evicted from the cache before they are needed.

ECMDeplanePage

eStreamStatus ECMDeplanePage(

IN **fileId** *File*,
IN **EStreamPageNumber** *Page*,
IN **char** *Buffer[ESTREAM_PAGE_SIZE]*
);

ECMDeplanePage performs all the necessary actions for writing a page coming off the network into the cache and back to the EFSD. This consists of copying the page into the cache, remove all pending requests for this page from the in flight list, marking the page as clean/unlocked, and returning the page to the EFSD for each in flight request.

ECMReadPage

eStreamStatus ECMReadPage(
IN **fileId** *File*,
IN **EStreamPageNumber** *Page*,
IN **ECMRequest** **Request*
);

ECMReadPage performs all the necessary actions for attempting a page read from the cache. The cache is checked to see if it contains the page; if so, the page is copied to the buffer, the EPF is notified of the hit, and appropriate status is returned. Otherwise, this page is put on the queue for requests pending to the prefetching component, and appropriate status is returned.

ECMWritePage

eStreamStatus ECMWritePage(
IN **fileId** *File*,
IN **EStreamPageNumber** *Page*,
IN **ECMRequest** **Request*
);

ECMWritePage performs all the necessary actions for attempting to write a page in the cache. Note that this could be somewhat more complex than a read, because a partial write to a page might necessitate reading the page from the server before writing the partial page to the cache.

The following interfaces are the abstract interfaces that the ECM will use to communicate with the EFSD. Hiding the EFSD's raw DeviceIoControls behind these interfaces will help make porting the ECM into the kernel easier, should we decide to do that.

ECMSetTokenState

eStreamStatus ECMSetTokenState(
IN **AppId** *App*,
IN **AppTokenState** *State*
);

ECMSetTokenState is called by the LSM to indicate to the ECM that a token has become available or has expired. The main effect of this interface is to update the state of the specified application in the active app table. See the ECM-LSM interaction below for more details.

ECMGetCacheInfo

eStreamStatus ECMGetCacheInfo(
OUT UNICODE_STRING *Location*,
OUT uint32 **CurrentSize*,
OUT uint32 **MaximumSize*
);

ECMGetCacheInfo is called by the client user interface to find out where the ECM cache is located and its current and maximum size. Location is an absolute path name of the cache file.

ECMSetCacheInfo

eStreamStatus ECMSetCacheInfo(
IN UNICODE_STRING *Location*,
IN uint32 *MaximumSize*
);

ECMSetCacheInfo is called by the user interface when a new cache location or size has been requested. Note that the cache manager may only begin using the new cache information after a restart of the client software (which may only occur on client machine reboot.) The client UI will call this interface when it wants to make a change; the ECM is responsible for actually resizing the cache and making any changes necessary to persistent storage (i.e. the registry).

EFSDGetRequest

eStreamStatus EFSDGetRequest(
OUT EStreamRequest ***Request*
);

EFSDGetRequest reads the next request from the EFSD, including any parameters that need to be passed. This may involve one or more DeviceIoControl calls to the EFSD. **EFSDGetNextRequest** is responsible for allocating memory for this request, and an **EFSDCompleteRequest** call will be repsonsible for deallocating the memory.

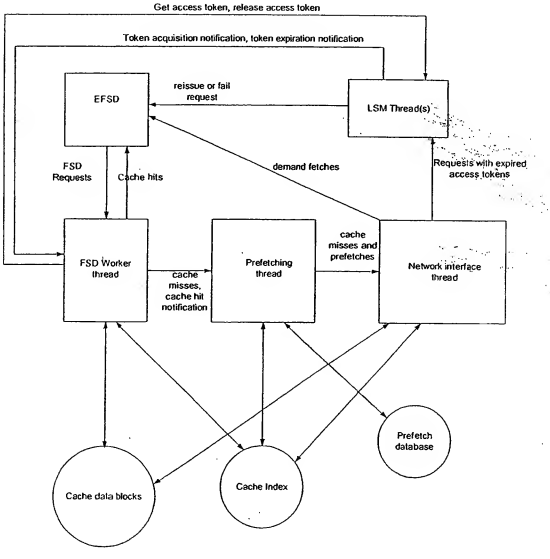
EFSDCompleteRequest

eStreamStatus EFSDCompleteRequest(
IN EStreamRequest **Request*,
IN ECMErrorCode *Status*
);

EFSDCompleteRequest will be called for each request that is received by the ECM via EFSDGetRequest. *status* indicates the completion status for this request, and may indicate success, a retry, or a particular failure condition. Non-persistent errors will be handled by the ECM internally or by requesting a retry of a particular request. Errors reported to the EFSD will be propagated up the file system stack.

Overall Client Architecture

The eStream client will have various types of threads in order to perform its work. The basic architecture is illustrated by the following diagram.



The FSD worker thread will pull requests from the FSD. It will return data for requests that can be satisfied immediately. Any request that requires information that is not currently in the cache will be put on a queue for the prefetching thread to handle.

The profiler will receive all cache misses from the FSD worker thread. Using its own data structures (which may include information about recent cache misses in addition to information about general prefetch patterns), it will decide which blocks it should prefetch. Demand fetch and prefetch requests are sent to the network component. The

only way demand fetches and prefetches are treated differently by the network component is that demand fetches are sent to the EFSD while prefetches are not.

The network thread will manage open connections to app servers and retry requests that time out. When data comes back from the network, the network thread will copy the returned buffer into the cache and to the FSD, if the request was a demand miss.

The cache manager consists of the EFSD worker thread and the APIs to access the cache index, the data blocks, and various queues used by threads in the client.

Not shown on the diagram is an error thread. This thread is responsible for calling the client UI module indicating appropriate error messages and waiting for the user's input. When any component decides that it has an error condition that requires user input, it calls **ECMReportError** with the request and an appropriate error condition, which will be enqueued for the error thread to handle. For example, when the network interface times out reading a page from an application server enough times, it will call **ECMReportError**. When the error thread gets to this request in the queue, it will ask the user if he wants to wait until the app server is available or allow the application to terminate.

ECM-LSM Interaction

The ECM-LSM interaction is a relatively simple one. The LSM notifies the ECM when it first receives an access token and when its access token expires. It does this via the **ECMSetTokenState** interface. The ECM keeps track of each application that has had files open, and whether or not we have an access token for each of these apps.

App ID	# of open files	Have access token?

Note that the LSM need not notify the ECM of mundane events like renewals as long as some token is valid. Also, the ECM does not keep track of the token itself, just whether or not we have a valid one. An additional nicety of this approach is that we could allow the ECM to satisfy requests out of the cache as if we have an access token, without actually having one.

When it receives a request, the ECM checks its table to determine if an access token is available. If it is, it handles the request as normal. If not, it asks the LSM to acquire an access token via **LSMGetAccessToken**. The LSM may return that it has a token, in which case the ECM will continue to process the request, or the LSM may say it doesn't have a token, in which case the LSM takes ownership of the request and will reissue the request when the access token is available.

When the number of open files drops from 1 to 0, the ECM will mark the token as invalid in its table and call **LSMReleaseToken**. The LSM may choose not to renew access tokens that have been released.

Component design

Two cache organizations will be presented. One is suitable for a quick implementation but doesn't lend itself particularly well to high performance or easy manageability; the other will be more difficult to implement but should provide better performance. I will first describe some data structures that are shared by both designs, then go into the specifics of each design.

Common Data Structures and Algorithms

Certain request lists are common to both cache organizations. One is a queue between the FSD worker thread and the prefetching thread for demand fetches that have not yet been seen by the prefetcher. The other is a list of all requests for pages that are "in flight." Requests from the in flight list are removed when they have been satisfied. The in flight list is unsorted and searched whenever a request comes back for requests that match the returned page. If the performance of this data structure becomes an issue, we will change its organization for faster lookup.

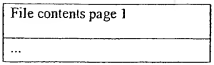
Both request lists use the request data structure described above.

The ECM will maintain an array of files currently opened by the EFSD. On file opens, an empty location in this table will be allocated for the newly opened file, and the index to that entry returned as the file handle. (Note that the way the interface between the ECM and the EFSD is defined, it is an error to open an already opened file. The cache manager will have to detect such cases and report an error, but it will not keep a reference count of the number of opens on each file.) This mechanism will allow the ECM to keep track of the volumes that currently have opened files as well as abstracting the client/server file ids away from the kernel driver. (This might allow us to update the client/server protocol without rewriting the EFSD.)

Easier Implementation

The cache will be implemented as a directory tree on the user's hard drive that parallels the eStream file system. Each file will contain a header and an array of status bytes in addition to the data blocks that the file contains. The array of status bytes has one byte for each page in the file. Each byte indicates the current status of that page in the file. (Pages have several different states, so a simple bit per page is not sufficient.) Each file will thus look like

Header
Page Status Bytes
File contents page 0



The header is defined as:

```
typedef struct
{
uint32 magicCookie;
uint32 headerLength; /* Length of this header, in bytes */
fileId fileid; /* for sanity checking */
uint32 length; /* Length of the file, in bytes */
uint32 firstPage; /* Offset to the first page in the file
*/
Metadata metadata;
} ECMCacheFileHeader;
```

The page status bytes begin immediately following the header, and this area is padded with zeros to a page boundary. The first page of the file's contents (and thus each following page of file contents) will therefore begin on a page boundary.

Note that one issue with this design is that files that approach the file size limit of the underlying file system cannot be represented, due to the overhead with the header and bitmap. If this design is used solely for early engineering efforts, then this limitation is acceptable. If we have to work around this limitation, one way to do it is to make the headers and page status bytes reside in a separate file or files.

Directory contents would reside in server format in a file named "Directory" inside of the directory whose contents they represent (with the addition of the header and status bytes as described above for ordinary files). For example, z:\Program Files\Microsoft Office would reside in c:\Cache\Program Files\Microsoft Office\Directory. This has the drawback of creating special file names that can't be used by files in the eStream volume, but again, for an early engineering implementation, this is an acceptable limitation.

Another issue with deploying this implementation is that it is trivial to reverse-engineer this file format and copy files directly from the cache.

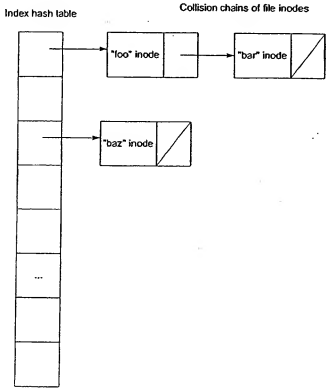
Robust Implementation

The cache will be organized into an index file and one or more cache data files. Multiple data files may be necessary as we may wish to allow the cache to grow larger than the 2 GB file size limit (for some native file systems) or to span multiple drive letters on the client. The data files will only contain pages of file content. These pages will be aligned on page boundaries. The index file contains all the information needed to locate file pages, and is contained in a separate file for simplicity.

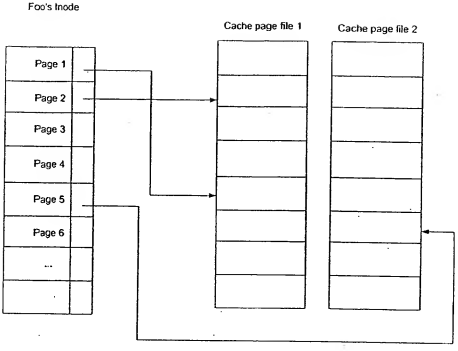
Page and index files must reside on a local disk (rather than a network disk) and cannot be shared by multiple clients.

Each file with any pages currently resident in cache will have a data structure containing information about that file, including its file id, the file id of the directory containing it, the file's metadata, and the map for finding the file's data blocks. This data structure is very similar to the inode of a traditional file system, and will be referred to as the eStream inode. A naive implementation of the inode is described above; no doubt, we will want to reorganize this data structure for more compact representation and better performance. Note that one requirement of the inode is that it contain a status field for each page in the file. One character is sufficient for this status; whether or not we can make do with fewer than 8 bits is an open question.

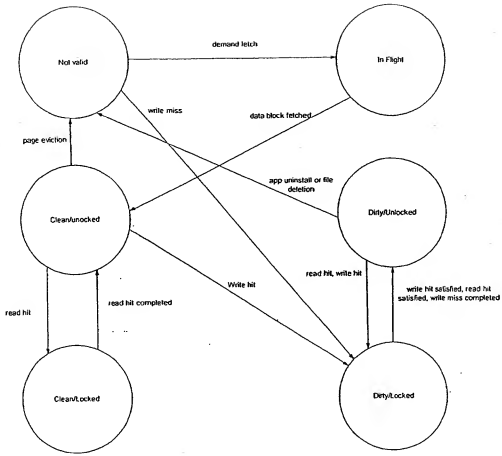
A hash table will be used to map file IDs to file inodes.



The inode contains pointers to each block's location in one or more cache page files:



To prevent race conditions, a single lock controls access to both the hash index and the linked list of requests that are pending network access. Individual pages in the cache may be locked for read or write access. Since each page's status is in the index, the index must be locked order to lock a page for reading or writing. The page states are controlled by the following state machine:



The dirty/clean distinction is between those pages that we have written locally (and thus cannot evict from the cache) and those pages that we haven't written (and thus can be refetched from the server).

A page would be locked while it was being read or written for copying to the file system driver. The operation may thus proceed with the index unlocked, without the possibility of page eviction while a copy is still in progress. The FSD worker thread is the only thread that reads or writes pages from the cache, so it's the only thread that can lock or unlock these pages. The in flight state is only for pages that are currently being fetched, either as a demand fetch or as a prefetch. The prefetching thread is the only thread that will put pages into this state, and only the networking thread will move pages from in flight to unlocked.

A list will be maintained of all "in-flight" requests. A single lock will control access to both this list and the cache index, so there are no race conditions between items being put on this list and data coming off the network. When the FSD worker thread gets a request, it acquires the index lock and looks at the status of the page. If the page is clean or dirty but unlocked, it will lock the page and copy it to the FSD. If the page is invalid, then this is a demand fetch, and the request is forwarded to the prefetcher. If the page is marked in

flight, then this is either a second request for an outstanding demand fetch, or it is a request for an in flight page. Either way, while this thread still holds the index lock, this request will be inserted into the list of in-flight requests. Race conditions might occur because the FSD might make multiple demand reads of the same page, or it may make a demand read to a page that is already in flight due to a prefetch.

Reading requested pages off the network and writing them to the cache (and to the file system driver, if necessary), are where this race condition comes up. We need to ensure that a request for a page that has arrived does not end up in the list of "in flight" requests. The solution is the following: When a data page comes back from the server, the networking component acquires the index lock to find the cache location of this incoming page. If the page is not marked in flight in the cache, this is a bug. (Of course, this is a relatively benign bug, and the NW component could just ignore the page.) The networking thread leaves the page as marked in flight, however, and unlocks the index. It writes the incoming page into the proper location, but it saves the in-memory copy of the page. It then reacquires the index lock, marks the page as clean/unlocked (since it's now in its final location in the cache), removes each request in the in-flight list for this page, then releases the lock. (Any further requests for the same page will find the page clean/unlocked, so the FSD worker thread will be able to satisfy these requests directly.) The networking component then proceeds to satisfy all of the requests it pulled off the in-flight list by using the copy of the page that it saved in memory. This way, it doesn't have to lock the index the entire time it is sending completed requests to the FSD.

Each of these complex scenarios is captured in the cache file's API's. As long as these are implemented correctly, other components don't need to worry about the exact sequence of operations that needs to occur.

Free Space Management

Free pages will be maintained as a free list in memory and as a bitmap on disk. The free list will be built from the bitmap on eStream client software startup. Access to the free list will be controlled by the same lock controlling access to the index.

Evicting Cache Pages

Individual cache pages may be evicted. There is an 8-bit field in the index for each page's importance. Initially, we will implement a random page replacement policy. Later, we will use this page importance field in an unspecified way to replace pages in such a way as to maximize interactive user performance and minimize application server load. Only clean/unlocked pages may be evicted. Pages that are evicted will eventually be put on the free list. Page eviction will only happen at "garbage collection" time. See "crash resilience and garbage collection," below.

Handling Cache Size

Growing the cache should not be an issue. The cache manipulation routines must know the overall size of the cache, in pages. Increasing the size of the cache on the fly should be a relatively straightforward process, as we merely need to lengthen the cache file(s) and add the new pages to the free page list.

Unfortunately, shrinking the cache is a much more difficult operation, since it potentially involves moving around pages that might currently be in use for paging operations or be in flight from the network. Changing the cache around at runtime is both difficult to implement correctly and a performance problem. The current plan is to support shrinking the cache only at eStream client software startup. The maximum allowed size of the cache will be stored in the Registry. On eStream client software startup, the current size of the cache will be compared against the allowed size specified in the registry; if it is larger than the maximum size specified in the registry, then the size of the cache will be reduced by evicting files and compacting the freed space. A request by the user to reduce the size of the cache will take effect the next time the client software starts.

Note that files that the user writes to the z: drive are not considered candidates for eviction (unless the file is explicitly deleted.) This means that the user's on-disk cache may in fact grow to be larger than the limit they specify.

Also note that at least one free page (not used by user-written files) is required for the file system to make forward progress. We also may want to require some minimal amount of cache before eStream will even run. Thus the maximum cache size specified by the user should be considered a "soft limit." There would be a "hard" minimum amount of space equal to the number of pages required to store the files written by the user on the z: drive plus a small amount of cache we designate just for running eStream. If this hard minimum is greater than the soft maximum specified by the user, the hard minimum would win. I would recommend preallocating and non-zero filling the file on disk so that we know that the space is available.

Crash Resilience and Garbage Collection

In order to provide crash resilience, the index will be periodically checkpointed to disk. Note that allocating blocks does not cause problems if the index is not updated. However, we cannot reuse a page's storage until that page has been marked free on disk.

The solution to this problem is to periodically garbage-collect the cache (if it is nearly full), and writing the index to disk. The cache manager will alternate between writing two cache index files. The index file will have a marker at the end that indicates that it has been successfully written and a time stamp, and on startup the ECM will use the latest, fully written index.

Data blocks will always be written directly to the cache page files. These files must be flushed before writing the index.

Garbage collection involves the following steps:
- lock the index
- copy the free list
- choose blocks in the cache to free, and make a list containing just the newly freed blocks. Mark these blocks as invalid in the file's inodes, but don't put them on the free list (yet)

- make a copy of the index
- unlock the index
- merge the list of newly freed blocks with the copy of the free list
- flush all cache page files
- write the new, merged free list (as a bitmap) and index to disk
- lock the index
- add the newly freed blocks to the free list
- unlock the index
- free any allocated data structures

Index File Contents
The index file contains the following items:
- List of cache block files, with their sizes
- Free block bitmap, per cache block file
- Inodes for all files; may be stored hashed or may be rehashed on startup.

Testing design

Unit testing plans

Cache file manipulation routines can be tested in isolation. We will write a standalone harness that exercises the functionality of the cache file manipulation routines by performing cache level operations directly. A multithreaded unit test for the cache manipulation routines would be ideal, so we can test the correctness and performance of our locking strategy without the need to build the entire cache manager.

Each "thread" of execution described by this document can be separately tested by creating a testing harness providing that thread inputs and monitoring its outputs. Replacements for the EFSD interfaces can be very effective here.

Stress testing plans

An interesting stress test for the cache manager is if it can work correctly with very small caches, even all the way down to 1 page. (Or at least, a cache with all pages but one marked as dirty.)

The cache manager will be able to operate in "verify mode," where requests that hit in the cache will still be sent to the server, and the pages returned by the server will be compared with the cached page's contents.

The cache manager will support multiple different page checksum algorithms. We can use a fast algorithm for deployment while using a more rigorous one in development. This also has the benefit of allowing us to test the performance impact of various checksum algorithms.

The cache manager will have the ability to verify the integrity of the cache index and free page bitmap. In particular, it will have the ability to determine taht no pages are allocated to more than one file in the file system, and that each page belongs to a file or is on the free list.

Stress testing for the ECM will include crash testing.

Cache manager testing will include resizing the cache.

Coverage testing plans

Cross-component testing plans

We can build a "cache only" file system by not using the prefetching and network components. This allows us to test the EFSD in conjunction with the cache manager without involving the prefetcher or the network component.

Early implementation of the client will likely involve a null prefetcher that does no prefetching.

We can use the testing harness for the cache manager that doesn't use the EFSD to drive the cache manager in conjunction with the prefetcher and network component. This allows us to test the combination of these components without driving it with the live file system driver.

Upgrading/Supportability/Deployment design

The client user-mode software and device drivers are packaged separately. (I.e. the client executable and the drivers are separate files on the disk.) This leads to the possibility of a "partial" upgrade that results in inconsistent versions of the drivers and client user-mode software. The drivers should support an interface that returns the version number of the driver, or of the interfaces provided by the driver. This will help the client software to recognize situations where it should tell the user to reinstall the client software and not result in bad system behavior.

Most (all?) on-disk data files should have file headers containing at a minimum a magic cookie and the file format version number. This will help us with upgrades in the future.

Open Issues

We need to address what happens when a fetch is requested and no empty space can be found in the cache. The prefetcher should probably block until such time as space is

made available for this request. While operating with very small amounts of cache will obviously cause bad performance, it should not result in a deadlock.